Masterarbeit

Design and Implementation of a Java Card Patching Mechanism for Smart Card Operating Systems

Franz Krainer

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Begutachter:	Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger
Betreuer:	Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger
	Dipl.-Ing. Johannes Loinig

Graz, im Oktober 2010

Kurzfassung

Eingebettete Systeme und insbesondere Smart Cards verfügen üblicherweise über ein Smart Card Betriebssystem und zusätzliche domänenspezifische Anwendungen im Read Only Memory (ROM). Daher ist *Patching*, die Korrektur von Software im Fall auftretendender Fehler oder veränderter Anforderungen, eine sehr schwierige Aufgabe, da eine einfache Ersetzung von Code nicht durchführbar ist. Zudem ist das Betriebssystem oftmals in mehreren Programmiersprachen verfasst und in Assembler-, C- und Java-Layer eingeteilt. Zur Abarbeitung des Java-Layers wird eine funktional eingeschränkte Java Virtual Machine als integraler Systembestandteil eingesetzt. Das Patchen dieses Layers gestaltet sich besonders schwierig.

Ziel dieser Masterarbeit ist der Entwurf und die Implementierung eines umfassenden Java Card Patching Mechanismus zur Veränderung des Java-Layers eines Smart Card Betriebssystems im ROM. Der eingesetzte Mechanismus ist äußerst flexibel und feingranular und erlaubt das Patchen einzelner Instruktionen. Die präsentierte Lösung ermöglicht hochperformantes Patching durch den Einsatz geeigneter Hardwaremechanismen.

Abstract

Small embedded systems and smart cards in particular usually have their smart card operating system (SCOS) and additional domain-specific applications stored in read-only memory (ROM). Therefore patching of the SCOS, in case of erroneous behavior or changed requirements, is a nontrivial problem because common code substitution is not feasible. Moreover the SCOS is often written in different programming languages and has different system layers in assembler, C, and Java. A limited Java virtual machine as an integral part of the system deals with the Java layer. Patching this layer is a particularly sophisticated task.

The main goal of this master's thesis is the design and implementation of a comprehensive Java Card patching mechanism for alteration of the Java layer of an SCOS stored in ROM. The mechanism is highly flexible and fine-grained in terms of replaceable modules. Single instructions can be patched. Furthermore, the presented approach enables high-performance patching through suitable platform mechanisms and full hardware support.

STATUTORY DECLARATION

Acknowledgment

This master's thesis was created at the Institute of Technical Informatics, Graz University of Technology in cooperation with NXP Semiconductors Austria GmbH Styria in Gratkorn.

First of all I want to thank my advisor Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger for the professional and organizational support in the course of the creation of my master's thesis. The excellent contacts of the Institute of Technical Informatics to industry partners were facilitating the thesis work in the first place. Especially in this regard I want to thank all participating institute members.

Very big gratitude goes to my second advisor Dipl.-Ing. Johannes Loinig for his amazingly huge support. Without his assistance and advice a successful completion of my master's thesis would not have been possible. Especially in the beginning of the practical work where everything around was novel, complex and confusing, he sacrificed precious hours to shed light on the dark.

Furthermore I want to thank NXP Gratkorn for the excellent cooperation and great support. It was really a pleasure to get to know such a professional working environment including perfect facilities, ambient comforts, and excellent cooperative team members. Special thanks go to Dipl.-Ing. Dr. techn. Ernst Haselsteiner for taking the time for crucial and critical questions providing his comprehensive knowledge in the field of smart cards. Moreover I want to thank all my colleagues at NXP Gratkorn for supporting me in so many different things I was encountering once in a while.

Last but not least I want to thank my friends and family for every possible kind of support. With their sacrificing aid it was possible to pass the deepest valleys and hardest times and stay on the right path.

Graz, October 2010 Franz Krainer

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Small embedded systems have gained more and more importance in recent years. Many gadgets nowadays come with microcontrollers on-board comprising complex hardware and software. Microcontrollers usually consist of a processor plus additional modules all on one chip (system on a chip). These additional modules are usually volatile and non-volatile memory, analog or digital input and output, distinct co-processors, timers, etc. One of the non-volatile memory modules, as an integral part of the system, usually hosts the code to be executed on the core processor of the microcontroller. This code very often resides immutably in read-only memory (ROM) which means that the functionality is determined at chip production time.

Smart cards are a pervasive type of small embedded systems. Whereas memory cards are typically used for storing data including some dedicated logic in hardware, microprocessor cards are the one containing an embedded microcontroller able to handle very complex processes. The kind of smart cards this work deals with hereafter are processor cards. Smart cards usually have an operating system typically residing in ROM. The operating system is the central point of activity in terms of accessing any components on the microcontroller, caring about security issues, etc., similar to common PC operating systems.

Smart card operating system (SCOS) source code for smart cards is generally subject to many security regulations, heavy testing for functional and non-functional requirements, and certification processes. Nevertheless errors or non-compliance issues are detected in completed products and in-use ROM masks from time to time. Hence a mechanism to fix occurring problems after production is urgently needed because total code correctness (following a formal specification) cannot be guaranteed in all facets. It is a crucial step already in the design phase to have so-called patching mechanisms in place to handle code problems in the operation phase.

Nowadays software for modern smart cards is often not written by one development team only. That means that either the approach of writing user programs in native code is taken (while OS core is mostly written consistently by one coordinated team) where accurate interface specification, memory protection, etc. is obligatory, or the interfacing is done via the application of different, clearly detached programming languages: that is the deployment of Java Card. On the one hand usual native code (compiled from C, assembler, etc.) is executed on the platform's microprocessor, on the other hand software for smart cards is written in Java and executed on a virtual machine (VM) which itself is a native on-card implementation. Separation of responsibilities and a principal security concept can be implemented in a very strong way.

Because the Java Card concept is prevailing more and more and parts of a common Java Card OS are also written in Java itself it becomes interesting and important to be able to fix problems in Java Card OS code as well. That is exactly the point where Java Card Patching for smart cards becomes a substantial and crucial issue.

1.1 Motivation

In the following section introduction and motivation towards small embedded systems, especially smart cards and their reference to security, are given. Subsequently the importance of smart card operating systems and with it Java Card is emphasized. Finally the reason for the ROM masking approach in the smart card business and the principal need for patching is explained.

1.1.1 Small Embedded Systems

The increasingly heavy deployment of small embedded systems is one of the technical phenomena in the last two decades. Almost every technical device nowadays has a microcontroller hidden somewhere in the inner workings. But why is this necessary in principle and driven by the production companies?

One very important fact is the falling cost. As the bulk price for microcontrollers falls below one Euro per piece [Com10] the implementation of logic functionality in software is very interesting and attractive in terms of business and market. Implementing this functionality in pure hardware in terms of dedicated integrated circuits (IC) may be much more cost intensive in most of the cases nowadays.

Increased flexibility is also a very important issue. Since software has the inherent advantage over hardware to be changed more easily, altered software modules can influence the behavior of the device without increasing the production costs severely. Commercial derivatives where only parts of the principal resources are activated are for instance a big step towards flexibility achieved by software. Certainly the possible application of software patches or firmware updates contribute to the overall flexibility.

Growing complexity plays an important role in preferring embedded systems to dedicated hardware. Since the number of different modules (input and output) for the logic entities increases and the requirements become more challenging (inter-module communication, modularity, complex processing, etc.) embedded systems may handle this better or make this just viable. A facile example for increased complexity in terms of the principal device in consideration would be a washing machine nowadays and a few decades ago. The requirements of a current machine would be a timer, water check module, accurate temperature controller, miscellaneous programs with additional modifiers, etc. whereas in the past all these accommodations were not required or simply not available.

1.1.2 Smart Cards and Security

The term embedded system is often used very comprehensively. Especially concerning the main common characteristics (minimal costs, small space, energy and memory saving) smart cards can be considered as part of the embedded systems family. One main specialty is that in smart cards the microcontroller itself acts as the leading part and is not just the logic brain in the background controlling the outer gear.

The motivation for application of smart cards in modern life is clear and versatile. In terms of identification e-passport would be the first flagship representing the identity of the passport holder and authenticating against the legal authority at a passport control. A credit card is typically a device that performs user authentication via personal identification number (PIN) verification (card holder against card) and also authentication in terms of charging the bank account (card against terminal and uplink). A smart card as ticket is a perfect representative for smart card authorization. Because of the ticket points stored and protected on card one can have access to the means of transport or not. Summing up, the general main tasks of smart cards are identification, authentication, authorization, secure storage, and general encryption plus decryption.

The advantages of smart cards are soon obvious: applicatory secure storage, principal cryptographic security boost for sensitive processes, forgery and fraud protection, etc. Smart cards usually contain cryptographic engines for encryption and decryption tasks, respectively. Information is for instance stored encrypted on the card, content is sent as ciphertext over insecure channels, or decrypted to plaintext after being received fully encrypted.

1.1.3 SCOS and Java Card

In Section 1.1.1 the importance of smart cards was clearly highlighted. So what is required to create such a secure device enabling all the strict requirements and expectations? On the one hand protected and neatly designed secure hardware is essential (forgery proof, analysis resistance, etc.) on the other hand secure software may be much more interesting, important and error-prone as well. As with the familiar personal computers, smart cards also have an operating system or SCOS. The SCOS has similar tasks like resource administration, security enforcement, etc.

There are many different systems on the market. The most obvious common ground for all the overall systems may be that there are running user programs on the SCOS, possibly written by more than one customer. Seldom the whole product or solution is launched by one company only. The problem of separated responsibilities can be solved via different operation modes like SCOS mode and user application mode (restricted memory areas and access rights) or via the application of a virtual machine where user programs are executed securely and separated from each other. This is exactly the approach Java Card [Ora10b] is heading for.

The main motivations for Java Card are portability, convenience, and security. To come to portability it is a big issue that one and the same applet (Java application for Java Card) can be run on different Java Card OS. This is one very important point for the SCOS user to be able to reuse code on different platforms (costs, time to market, etc.). Programming in Java is also simple, convenient and supported by numerous tools being an industry standard for many years now. It is much easier to program in Java than maybe in any proprietary assembler or similar. Code is reusable, modular, object oriented and has the potential to be written and read understandingly, depending on the developer as well. When it comes to different SCOS and user applications for it, inherent security mechanisms are the most important arguments towards Java Card: intrinsic Java data security concepts and the execution on a sandbox-like virtual machine, applet firewall protecting different Java Card applications from each other, and supported standard security mechanisms. Java Card Technology is - because of all the issues mentioned above - breaking new ground by typifying security, robustness, and small device applicability by design and concept more than any other technology on the smart card sector.

1.1.4 ROM Masking

One of the main incentives and cause for patching on smart cards, as described in the following, is the underlying immutable SCOS code in ROM. But what are the main motivations for ROM masking of SCOS?

Low cost for a huge amount of chips is the main criterion arguing for ROM masking. It is the smallest of all memory types and thus cheaper to produce once the data mask for production is created out of the current ROM image [Smi10]. Fast access and energy saving are also very important for ROM application on smart cards. Because in most cases ROM is quite simply built-on in principle it may also guarantee fast access times and optimality in energy saving. From the security point of view a SCOS residing in ROM is also well suitable. The permanent change of code through aimed attacks is non-feasible because a change of mask ROM is simply physically not possible in contrast to a SCOS in any mutable memory, e.g. electrically erasable programmable read-only memory (EEPROM) or Flash memory. Data retention and data safety are optimal and can only be improved by advancing the ROM production process. Failures may happen in the masking process of course.

All things considered it is a logical consequence for small embedded systems to place the SCOS in ROM at production time as it is cost saving, fast, secure, and energy saving. This is especially true for smart cards where huge amounts of cheap chips shall be issued.

1.1.5 Patching

We have seen the necessity for smart cards, the advantages of Java Card in modern environment, and the foundation in terms of ROM masking for SCOS. The most interesting issue is still the need for patching in principle.

Typically time to market is very important in a fast moving business like the smart card business. That means that the time to a ready-made image for an SCOS is very limited. Formal proofs over SCOS source code following an accurate formal specification are very hard to apply and hardly feasible. So the code can never be considered as fully flawless at production time and potential errors or non-compliances to industry standards are accepted to a certain extent. A technical backdoor has to be left open to change the SCOS behavior after chip production: patching. The mechanisms may be diverse and different (depending on SCOS location and configuration, chip characteristics, etc.) but the principal goal remains the same.

Provision of fast fixes or even prototypes (additional functionality differing from correcting errors) can be reached by the appropriate application of patches. The crucial issue is the balance between a new ROM mask and a fix via patching. When the focus is on the swiftness of an alteration then a patch is often favored even if there was time to stop the actual masking process and the product is not released yet. Still it is much faster to go the patching way because starting new ROM masks is very time consuming. The patch can be applied after the chip left the primal production stage (more in Section 6.5).

New ROM masks (as a consequence of unexpected malpractice) are preferred whenever the change is really big or it is technically infeasible to apply a patch. It cannot be considered as a fully coequal alternative to patching because if a load of chips is already out of factory or even in field operation patching is very often the only possible option. Every other approach would imply the exchange of the device itself.

To sum up, it can be stated that errors are as ubiquitous as smart cards are in modern life. That means that mistakes may appear everywhere in source code. For a Java Card OS - where parts of it are written in Java itself - these errors may appear in native and Java code. The result of compilation and the code in ROM is hence machine code and Java Card bytecode. It is definitely established to patch machine code but what happens when SCOS bytecode in ROM or any built-in Java Card applet ROM bytecode is erroneous? Therefore a Java Card patching mechanism is needed urgently, bytecode executed on the Java Card virtual machine shall be patchable and a Java Card application has to be alterable in terms of behavior.

1.2 Outline

This section shall give a short overview of the following chapters in this work to offer a guidance for the interested reader.

In chapter Basics the underlying technologies shall be explained and specified properly to get to know the basic terms and the field of application. Smart cards as vital technical object, smart card OS as central point of activity (including prevailing standards and technologies), and the predominant term patching shall be specified.

In chapter Related Work different approaches towards patching - especially in the embedded system sector - are examined. Differences and commonalities shall be elaborated.

In chapter Design of the Java Card Patching Mechanism the principal problem description shall be given, patchable elements will be identified, requirements in all facets are collected and specified, and subsequently the design is modeled clearly in terms of evolving a patching model for Java Card. The most important and essential part of the work, the Java Card patching core mechanism is illustrated in Figure 4.13.

In chapter Implementation of the Java Card Patching Mechanism first the development environment is described, then a realization overview shall be given showing the principal enabling elements. Subsequently the implementation of the patching mechanism is examined and characterized: patch generation, setup and initialization, and patch execution are the different main sections.

Chapter Experimental Results and Discussion is collecting the findings and outcome of the preceding chapters. A qualitative and quantitative evaluation is given showing performance issues in terms of time and memory.

Chapter Conclusion and Future Work eventually sums up the master's thesis and gives an overview of new perceptions and insights. Some future prospects on improvements of the overall patching mechanism are given.

Chapter 2

Basics

This chapter is about all the basics around smart cards and the corresponding technologies. First the term smart card will be defined giving a comprehensive overview of different classifications, typical characteristics and the most important specifications around. The landscape of operating systems on smart cards shall be discussed and specific differences are elaborated. Afterwards, the focus shall be on the Java Card Technology explaining the basic ideas and miscellaneous specifications. The final section deals with patching. First of all the term and concept of patching is examined. It is attempted to find classifications for different types of patching. In the following Java Card patching is defined and introduced.

The subsequent sections are based on renowned smart card literature. Basic definitions and concepts are taken out of it with regards to the contents [Che00], [RE03], [Hen01]. Special references and quotations, in particular in the field of patching, are highlighted individually.

2.1 Smart Card

A smart card is a mostly pocket-format plastic card hosting integrated circuits embedded in the surrounding body. It is also often referred to as chip card or integrated circuit card (ICC). As mentioned in Chapter 1 smart cards can be seen as part of the embedded systems family especially when it comes to typical characteristics. The main properties are: limited memory, limited computational resources, low costs and high production amounts, strong environmental influences, and hard application requirements. All the main characteristics become manifest in deployed smart card implementations in different areas. ROM is used for the basic code of the SCOS because of space and cost issues next to a limited amount of volatile and non-volatile alterable memory. Low-cost 8-bit microcontrollers are typical for smart cards, especially 8051 microcontrollers and derivatives are very popular [Aya04]. Smart cards use flexible hard-to-break carrier materials because of the harsh environmental conditions. The main requirement is to shield the chip from damage and destruction. Security is one of the main issues in smart card application. Thus basic security mechanisms are included in every smart card design step: protection against forgery, data theft or re-engineering through unauthorized system inspection, and generally any form of misapplication.

In the field of smart cards and smart card environments, there are typically many different manufacturers and operators involved. Different smart card types have to work in combination with different card acceptance devices in various configurations. So the existence of engineer standards and proper specifications is essential. The most important are outlined in the following. ISO/IEC 7810 [ISO10] is dealing with physical card formats whereas ISO/IEC 7816 [ISO10], [Car10] deals with various properties of contact cards. ISO/IEC 7816-4 is in particular about the transmission medium independent interindustry commands at application layer. ISO/IEC 14443 [ISO10] specifies contactless smart cards. Very important further standards are the EMV specifications [EMV10] and the Global Platform specifications [Glo10]. The EMV specifications constitute a global standard for credit and automated teller machine (ATM) cards based on chip card technology. It is driven by the most important card issuers. The Global Platform specifications are covering the entire smart card infrastructure from cards to card acceptance devices and systems. They form an open standard driven by a committee open to any interested organization.

There are several possible classifications for smart cards: by card size, internal characteristics, communication interface, and by application area. Based on these criteria Figure 2.1 illustrates a possible classification for smart cards.

2.1.1 Classification by Card Size

Smart cards exist in different sizes and forms. This ranges from the typical credit card format, to a special former format for German identity cards as well as a distinct format for passports and visas, up to a mini format for subscriber identity module (SIM) cards or similar. All different styles are defined in ISO/IEC 7810. It can be stated that a standard credit card as well as a passport or a SIM card are smart cards of different physical appearance.

2.1.2 Classification by Internal Characteristics

Much more important than the physical characteristics are the internal properties for classifying smart cards in terms of their functionality. The main differentiation is by the degree of card capabilities and potential. As the first category, memory cards are often free of complex processes on card, they provide their memory for write and read operations. Some dedicated security functionalities may be on card as well to provide authorization methods for memory access.

The second category are microprocessor cards. Memory is not supposed to be accessed directly but always via the microprocessor. Any access without permission is denied and respective mechanisms are in place to enable adequate protection. Microprocessor cards are able to handle very complex processes in comparison to pure memory cards. As they have a microprocessor on board it is possible to run application specific programs. This feature enables great flexibility. Typical processor cards hold an SCOS in ROM which is in charge of the basic access mechanisms similar to any general purpose operating system.

2.1.3 Classification by Communication Interface

There are two basic ways of communicating with a smart card in terms of the physical communication interface. Either communication is done via contactless interface over the

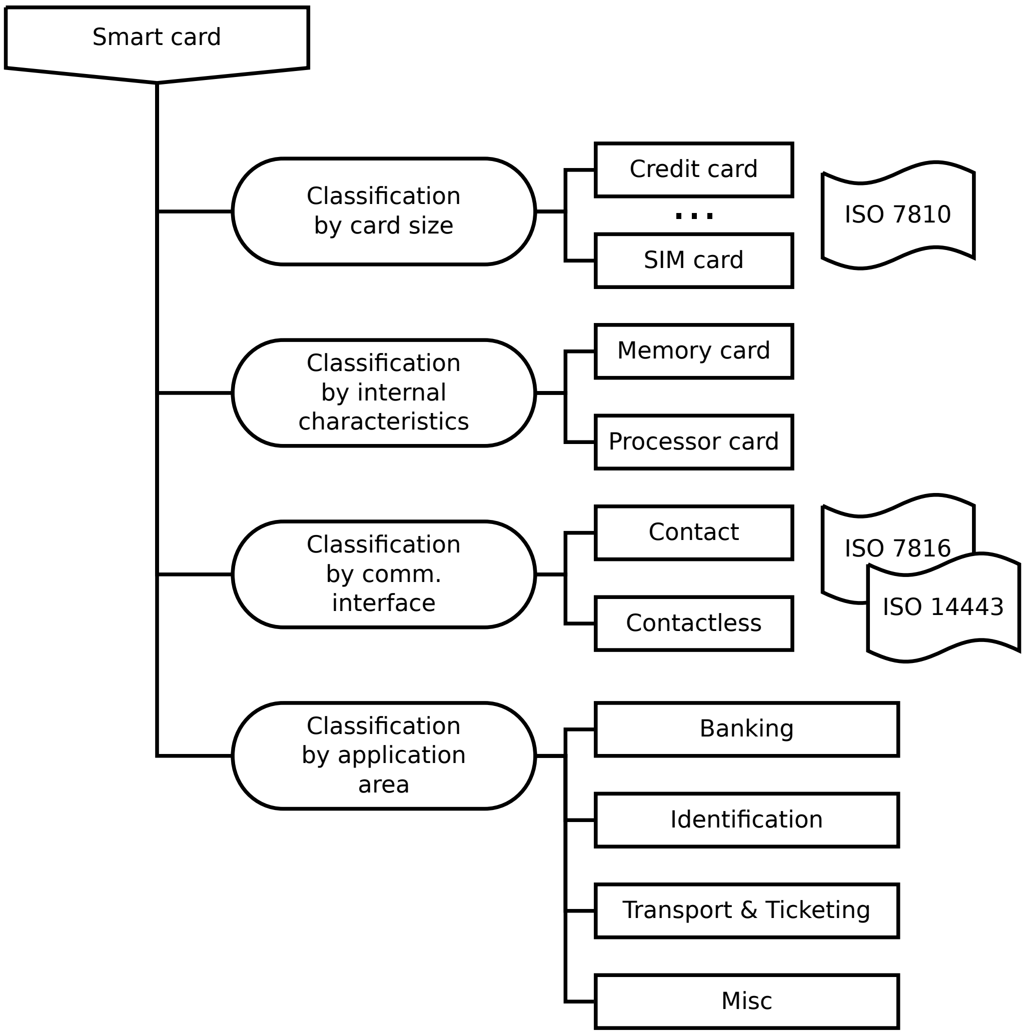

Figure 2.1: Classification for smart cards

air or via a metal contact interface directly connected to the chip. Some cards are able to handle both interfaces.

Contact smart cards are generally described in ISO/IEC 7816-1 to ISO/IEC 7816-3 from physical characteristics up to communication and interindustry commands (interface independent). For contact communication two important standard protocols are in use: T=0 and T=1 protocol defined by ISO/IEC 7816-3. While T=0 is byte oriented T=1 is a block oriented transmission protocol for contact communication.

For contactless communication the ISO/IEC 14443 standard is essential. Any characteristics for contactless cards are described. ISO/IEC 14443-4 defines the state of the art contactless transmission protocol T=CL.

2.1.4 Classification by Application Area

Smart cards may be used in very different environments and under various circumstances for many different purposes. To find a satisfactory classification it may become necessary to collect the commonalities and investigate the boundaries.

One main application field is the banking and financial sector. Credit cards or ATM cards are typical examples for smart cards. Card holder authentication is done by verification of the owner PIN and the principal smart card ownership. The card holder is then authorized to gain access to his bank account. Electronic cash (e.g., MasterCard Quick Payment Service) is also one of the possible applications.

Identification and especially e-government is another important sector comprising e-passports, citizen cards, smart health cards, and driving license cards. Main uses include identification of the card holder, authentication to legal authorities, and even secure storage of sensitive information (e.g., possibly for health cards). Next to e-government there are some further applications: company site access cards are a growing business because enterprise security and protection of intellectual properties are gaining more and more importance. The common factors are identification and authorization.

Another fundamental application area is the area of transport and ticketing. Public transport is the main player where millions of pieces of smart cards are sold. For this field it may become very interesting to grade the available security levels and provide different kind of smart cards from low security throw-away cards for short term up to high security for valuable long-term cards.

Other sectors are rather small in comparison but yet not unimportant. Cryptographic access smart cards like decoder cards are used in television business. Furthermore smart cards may be used for computer security as well: protection of keys for encryption, storage of certificates for secure browsing, or single-sign-on solutions are some popular examples. Forgery protection is going to be an interesting field as well as forged products are a big issue for many companies and the black market is prospering. Smart cards may enable mutual authentication of products or principal identification of a product as a proper original.

2.2 Smart Card Operating System

In the last section one important classification was memory cards versus processor cards. In the presented approach processor cards are in use and in the following the term smart card is identical to processor card. One main characteristic for this kind of smart cards is the existence of an SCOS. This becomes necessary for accessing any platform resources and peripherals, in particular file management, security, I/O, access to special co-processors, etc. The world of smart card operating systems is huge and diverse and so are the different concepts. In the following sections a short overview of the SCOS landscape is given and different approaches are examined. Special focus is then on Java Card OS and the Java Card technology.

2.2.1 General Considerations

An SCOS is essential for a processor card as it is the underlying software for handling every complex or simple process. Even though it is sometimes difficult to compare SCOS to general purpose OS, there are points of intersection. On a general OS we can see (user) applications as programs running natively on the hardware processor under certain restrictions and under the OS constraints. Or applications may run on a virtual machine like the Java VM or the Common Language Infrastructure virtual machine for the .NET framework abstracting the underlying OS and making code platform independent.

On smart cards the situation is similar and maybe even more extensive. Smart card applications are often comprehensively seen as applications running on the smart card processor but off card as well. So first of all one can differentiate between fixed command and free programmable SCOS. In a fixed command SCOS only the operating system manufacturer is able to extend the SCOS and supports a fixed set of commands that can be sent to the card (mostly following ISO/IEC 7816-4 and above) from the off card environment.

In a free programmable SCOS the situation is different. While on the one hand the SCOS itself implements basic commands (mostly following ISO/IEC 7816-4, etc. as well) on the other hand it is supposed to be extended by means of a dedicated development environment. In that way specialization in various business areas can be accomplished. Seen from a technical point of view the extension can be conducted in many different ways. One approach is to separate user programs from the operating system via a memory management unit and the use of a dedicated restricted user mode. These programs are then existent in native code and mostly written in C or assembler. Another important approach is to separate user programs from operating system via the usage of a different programming language and the application of a virtual machine. This is exactly where the Java Card technology is applied. The idea and concept is explained in the following section.

2.2.2 Java Card Technology

Java Card technology enables resource-limited devices (as smart cards) to run small applications, so called applets, that employ Java technology [Ora10b]. Java Card technology aims to fulfill several business and technology goals. Interoperability is very important. That means that applications designed for Java Card shall run on different Java Card platforms regardless of the vendor or underlying hardware. Security is an absolute main feature. The card issuer as well as the (potentially different) application providers have a high security level. Neither the underlying platform nor the applications shall be compromised. Multi-application-capability is also essential. Secure co-existence of different applications on one single card is guaranteed. Flexibility is another main goal: It is required to be able to load applications on the card after the primary card issuance. The compatibility with important industry standards as ISO/IEC 7816, EMV, Global Platform, etc. is the final important property to guarantee industry compliance.

For development the main advantages lie in the usage of Java as programming language itself: object orientation and the associated re-usability, any protection mechanisms typical for Java (strong typing, array boundary protection, etc.), and inherent clarity of the Java syntax and style. Furthermore many off the shelf tools are readily available.

In short, the basic work flow concept behind the technology is as follows. User programs, so called Java Card applets, are written in Java regarding certain Java Card platform limitations. A general purpose Java compiler converts source code into class files and a converted applet (CAP) file converter converts the Java bytecode to Java Card bytecode and creates a special structure and format suitable for usage in a very memory limited environment. The resulting CAP file (JAR file comprising all CAP file components) contains all information out of exactly one Java package whereupon an applet component is optional. The CAP file is the one entity which can be uploaded to the smart card. Uploading means the physical placement of the CAP file in non-volatile memory on the device including the on-card registration of the module and preparations for further procedure. On card a Java Card runtime environment is existent comprising a virtual machine, an API, and additional runtime support services. After upload an applet can be installed which creates an applet instance, and selected which means that the applet receives any further commands. Deselection, deinstallation, and physical deletion are also possible. Eventually the respective Java Card applet bytecode can be executed on the virtual machine. The applet component is the one central element comparable with a class containing the main method. Multiple applets may be on one card but are separated from each other securely. Next to uploading of Java Card applets in non-volatile memory it is also technically feasible to place applets in ROM. The applet can already be installed or even selected from the beginning. The Java Card Technology relies basically on three specifications:

- *Java Card Runtime Environment Specification* defines the Java Card runtime behavior for Java Card technology. The general behavior and environment in terms of the virtual machine, API, and support services are defined.
- *Java Card Virtual Machine Specification* defines the Java Card virtual machine for Java Card technology. The virtual machine general behavior, the VM instruction set, supported subset of Java language, and different file formats are defined.
- *Java Card API Specification* defines the Java Card Application Programming Interface for Java Card technology required to support the Java Card virtual machine and the Java Card runtime environment.

2.3 Software Patching

Software patching is the central topic within this master's thesis. As already indicated in Chapter 1 software may be erroneous despite all precautionary measures to avoid malfunctions. For many business domains in IT it is not common to do formal proofs over programs to prove the correctness in terms of formal specifications. Errors or non-compliances may occur or the improvement and thus alteration of software might become necessary after the first release. There are many attempts to define the term patch properly. One is done in the following paragraph [Dai04]:

> A change to a program - usually to correct some error - that emphasizes convenience and speed of change rather than security, and is intended to effect only a temporary repair.

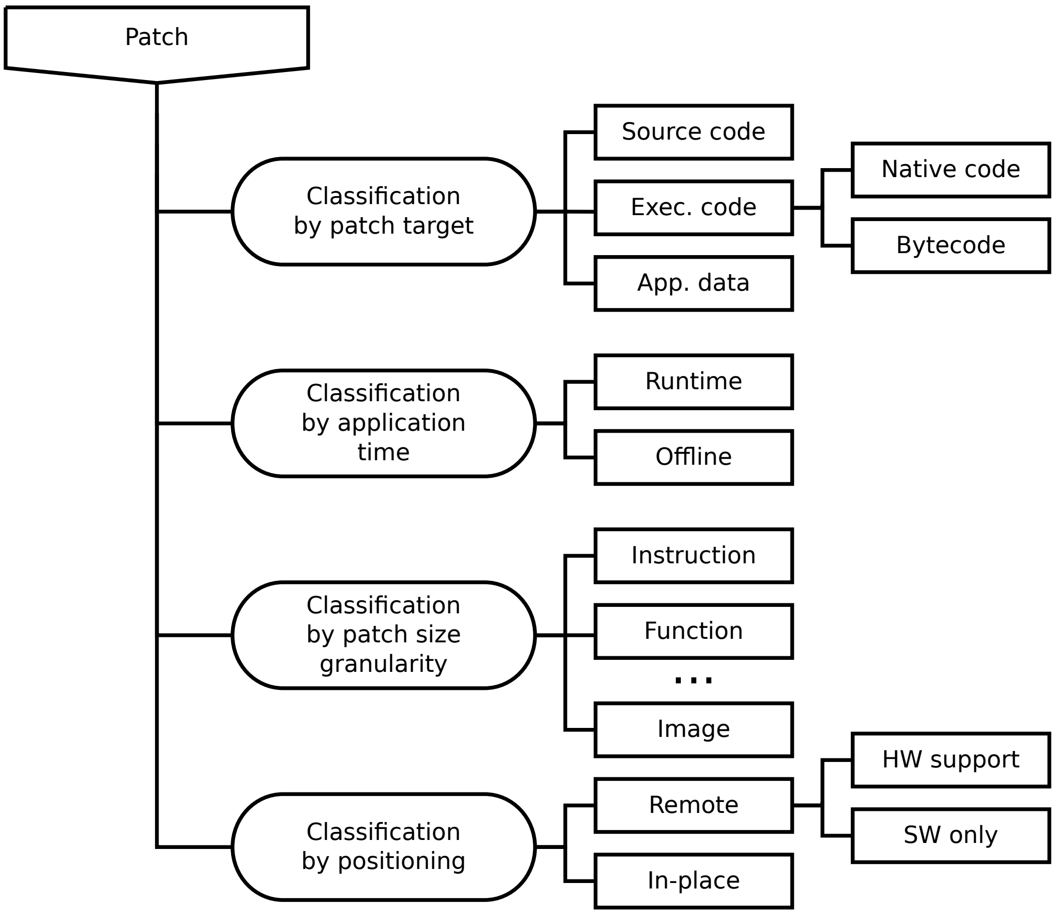
Patch
Classification by patch target
Source code
Exec. code
App. data
Native code
Bytecode
Classification by application time
Runtime
Offline
Classification by patch size granularity
Instruction
Function
...
Image
Classification by positioning
Remote
In-place
HW support
SW only

Figure 2.2: Classification for patches

A software patch can be understood in very different ways. The appearance of a real functional error for instance is not necessarily required. It may as well be the changes of environmental circumstances or similar that require changes in software. Depending on the software setting the temporary aspect may be more or less applicable. A patch can have persistent and temporal effect. The relevance of security is also a matter of the application and field. Rapid patch development and flexibility though is always a very important property as everything else would in the end lead to a totally new software release and no need of any patching activity.

In the following section an attempt to classify a patch and to find type clusters is done. It can be seen quite soon that there is a very broad range of different patch types and approaches to fix software.

2.3.1 Patch classification

The core meaning of the term patch is the change of software. There are so many ways of fixing applications depending on the overall platform setting, application field, and general requirements. Figure 2.2 illustrates one possible classification for patches.

A first basic issue is what really constitutes the target of the patch. Often files holding the difference between original and target source code are referred to as patches as they change source code to a new version. The other case concerns the patching of code that may be executed on a (virtual) machine. A third possibility is the patching of any configuration or arbitrary data that is used within a program. To get one level deeper the resulting code of a patch is interesting. Either it is native code compiled for a specific hardware platform or it is bytecode for any virtual machine. Exactly this case is very interesting in the course of this thesis.

A very important question to answer is if the patch has to be applied during program execution or not. If former is the case a patch is called hot patch or runtime patch otherwise cold patch. The difficulty to apply a patch during runtime may become enormous. The impact of patch application on a running program has to be considered carefully in terms of basic availability and responsiveness, timing constraints, security, etc. For cold patching things are much easier as all the runtime application considerations do not have to be made and fewer dynamic issues have to be considered.

That leads to the next possible classification: patch size granularity. A patch can be the replacement of a single native instruction or a few lines of code, an entire functional module (e.g., a function or class), a big process like module (e.g., dynamic linked library) or in the extreme case a complete program image.

Another level is the physical capability of positioning. While on the most platforms it may be possible to patch at the very location of error in terms of exchanging code (in-place), on some platforms that might not be possible (remote). This is especially true for most of the smart card where the SCOS is immutably existent in ROM. For the remote patching approach the classification could be refined. Hardware-supported versus pure software and to that effect trigger-based versus poll patching could be another reasonable categorization.

2.3.2 Native Patching

Generally native patching means the fixing of problems in applications compiled or translated for a specific hardware platform or microprocessor. The resulting executable native code of the application is the patching target and subject to any changes.

2.3.3 Java Card Patching

Generally Java Card patching means the fixing of problems in Java applications for the Java Card platform. For the presented approach in particular it concerns the correction of problems in Java Card bytecode and the Java Card application environment (Java Card stack, Java Card heap, and Java Card method area). When talking about Java Card patching many things and problems are totally equal to patching of a pure Java application on a Java virtual machine. It is more the environment and the special conditions that render Java Card patching unique. The proper classification of the presented Java Card patching approach in terms of Section 2.3.1 is given in Chapter 5.

It can be immediately seen that native patching and Java Card patching differ. While on the one hand the target native code is executed on a real microprocessor on the other hand Java Card bytecode is executed on a virtual machine which itself is running on a microprocessor. These facts lead to totally different conditions for patching. Java Card patching introduces different and mostly more difficult problems in realization.

Chapter 3

Related Work

Patching is a very broad topic and may be understood quite differently (more in Section 2.3.1). Indeed many patching mechanisms do not have so much in common except the basic idea of fixing a problem. In the specific area of trigger based patching and especially in the Java Card patching field of application not much related work exists. Many papers handle very specific topics or patching under dedicated circumstances, few of them seems to be comparable to the presented approach.

Java Card patching is a very difficile topic. Patching Java Card bytecode is nothing that is obviously necessary under usual conditions like on a personal computer or similar. It is moreover simply needless and far too complicated especially in systems where the replacement of entire code modules or simple re-compiling is not problematic at all. When it comes to smart cards and embedded systems where Java Card bytecode may reside in ROM everything changes completely.

3.1 Different Approaches and Patching Mechanisms

In the following sections a broad and comprehensive overview of different approaches towards patching shall be given. The application domains are varied and hardly comparable and it can be seen that patching covers a lot of different topics.

3.1.1 Runtime Application Patching for High Availability with Carrier-Grade Linux

In the online reference article [Meh10] a special kind of runtime application patching for high availability systems is characterized. The system used is a so-called Five Nines System which is defined by not having more than five minutes of down time per year (99.999% of uptime [HR04]). It is essential not to shut down the system, or, more generally, make it somehow unavailable at any time. Still it shall remain possibly mutable and thus patches have to be applicable. This requires a hot patching functionality instead of a cold patch or software upgrade resulting in a certain down time.

The patching mechanism is realized and implemented on a carrier-grade Linux. These linux systems are designed for telecommunication systems in the high reliability and availability sector, and are defined as follows:

> To enable VoIP traffic, application servers must provide carrier-grade reliability that guarantees high service availability (up-time of 99.999% or better). In addition, these systems must scale to handle hundreds of thousands of calls and provide predictable performance, and high speech quality [Meh10].

For this kind of system even the application of redundancy may not suffice. Fixing a bug on the replicated system respectively system module and emplacing it instead of the original one may also lead to down times and unexpected behavior because of synchronization issues (replicated part has to be brought down, patched and up again), etc. Still this would be a possible alternative to direct online patching.

The implementation of the patching mechanism in Linux is done via shared objects including special information. In the reviewed approach patches may introduce new methods and functions, add new ones, add data elements and so on The mechanism however cannot patch every possible data item (in terms of extending size, etc.) or infinite loops.

The patch mechanism consists of several stages: loading, activation, deactivation, and unloading. In the loading phase roughly the following is happening: dependencies between patches are resolved, all required patches are loaded into memory, some further preparations are performed but the patch is not activated or even executed yet. In the activation phase subsequent steps are worked off:

> When the patch is activated, the patching system suspend all threads, making sure they are not running in the areas of code being patched, inserts code at the beginning of the old functions that jumps to the new function code, and then resumes the suspended threads [Meh10].

There are also some supportive functions referring to activation and deactivation being executed before or after the entrance to the corresponding stages. Dependencies between patches are handled carefully and are monitored during loading and activation phases. Still, online patching as shown in this chapter may introduce a lot of problems that require intensive knowledge of how to create the patch and the original code: calling positions of patched functions have to be reconsidered, multi-threading is a big issue, etc.

3.1.2 Semantics-Preserving and Incremental Runtime-Patching of Real-Time Programs

Another very interesting approach is presented in [KLM09]. The work also fits into the area of patching hard real-time systems. Here programs describing non-functional aspects of processes are considered and patched at runtime. The programs are written in the Hierarchical Timing Language (HTL) and new code is placed during execution time. In the HTL programs things like timing properties and communication behavior are defined.

The terms semantics-preserving and incremental runtime-patching are central in the paper. Semantics-preserving means the modification of a program at runtime in a way that could have been done at compile time if the patch and the timing for it had already been known. Incremental here means that analysis and generation of the patch code needs an effort proportional to the patch code size and not to the program to be patched. For the actual patching a patch supervisor process is established, allowing loading, analyzing and applying of patches at runtime. In this approach the HTL code is re-compiled and linked at runtime including syntax validation and analysis of the schedulability. Only modified parts have to be handled and re-generated due to special code generation strategies.

In a nutshell, many issues have to be considered scheduling concurrently running tasks and the patch supervisor has to have the capability to apply transformations out of a set of patches at appropriate time instants.

3.1.3 Software Patching in the SPC Environment and its Impact on Switching System Reliability

A further very interesting work concerning high reliability and availability systems is presented in [Ali91]. The work highlights some principal concepts related to software patching (meta patching levels) especially targeted at stored program control (SPC) environments. SPC is the technical name used for telephone exchanges controlled by a computer program stored in the memory of the system [Man10]. What characterizes SPC systems are the real-time requirements with tens of thousands of I/O ports. Real-time is required inherently because the telephony environment demands it by definition. It is easily conceivable that patching such hard real-time systems is a task having high complexity.

The paper highlights (next to many technical issues) the various human interfaces and departmental responsibilities. Problem reports come straight from customers, the supplier produces a patch and a set of patches is then undergoing a sophisticated validation and distribution chain. It is stated as well that the application of the patches is in fact very complex and prone to human errors and especially critical to high-reliability systems. On a meta level the software change process is described, comprising seven different activities.

Request Comprehension is a very complicated and important stage. The whole process is usually triggered by a user request to fix an occurring problem in software. The most critical part is then the interfacing contact person (e.g. as part of customer account service, CAS) supposed to understand and formulate the problem. The so-called “user language” has to be familiar to the contact person who then communicates the problem to the developer core team (or maintenance software group).

The level of *Request Transformation* is then set off. Therein developers are required to transform the (formal) request into executable code. The impact of the requested change has to be best known and a deep understanding of the overall system is essential. Code changes are understood to be minimal so that software robustness is influenced as little as possible. Ideally code generation shall be accomplished by the same means and processes as the original software to guarantee uniform software quality.

In the *Patch Type Specification* stage two types of patches are distinguished: overlaid patch and patch-space patch. If the code size for the patch to be applied is quite small and the removal of old code is done, the patch may be overlaid and no additional patch space is needed. A patch-space patch though needs more and separate space, including a certain functionality to allow context saving, execution and return from patch. The terminology for this module is cut-line code.

The subsequent level is called *Software Change Testing.* The patch itself should function correctly and is tested in the usual development environment. Any side effects are very important to be investigated. Thus regression testing of directly involved modules and indirectly ones is essential.

The necessity for *Patch Documentation* is also highlighted in the paper. The patch purpose and impact, setup, and testing (field environment) shall be described properly to be able to track the changes in the affected software.

Patch Tracking is accounted an own important level: formal configuration and a mature control management system are demanded and integration failure report system shall be considered.

Patch Integration is the last level stating that patches are short time fixes only and changes shall flow in the source code of the next generation software product. All related software and documentation is supposed to be up to date.

Real-time systems usually have the problem of applying patches in conjunction with down times of the system. One possible strategy is the module exchange technique where modified software modules are simply swapped followed by a short reinitialization phase. The difficulty of the patching process also depends on the level of development or operation the device is currently facing. While in the coding and testing phase patching can be understood as fixing an error in source code and re-start standard development and testing procedures, in the operational phases patching becomes technically and even company-politically more explosive. As soon as the customer is involved in the patching process things begin to become critical and much more sophisticated. The negative impact of poorly designed patches is also considered. Excessive patching points to bad software quality or design, the risk of ending up in non-serviceable spaghetti code becomes high. Especially for highly reliable systems the amount of patches shall be minimized due to the fact of high malfunction risk and exploding costs. In the SPC field of application an embedded patcher concept is realized very often. The patching concept is already included in the design phase: validity check, sequence check (patch order), activation and deactivation control, backout control (potential removal), and listing control are the most important features included.

3.1.4 An API for Runtime Code Patching

The article [BH00] presents an Application Program Interface (API) for runtime code patching dealing with post-compiler program manipulation. The main goal is to manipulate and patch applications in software during runtime by means of a C++ class library. Patching is herein understood as a means to instrument and manipulate code in terms of debugging, logging or similar. Any activities are done and patched in at runtime.

The API presented permits a specialized program to connect to a running process and to influence respectively alter the execution code (mostly in terms of adding debugging or monitoring code). The main motivation in the paper is the patching of large, long-running programs like simulations which are not desired to be stopped, altered, re-compiled and re-started. Platform independence is very important as well, so that instrumentation code can be used on different platforms using a higher abstraction level. Abstractions for programs to be patched are a central topic in the API interface. The most important abstractions are *points* and *snippets*:

> A point is a location in a program where instrumentation can be inserted. A snippet is a representation of a bit of executable code to be inserted into a program at a point [BH00].

In fact, for monitoring a function or similar, the point would be the start of the

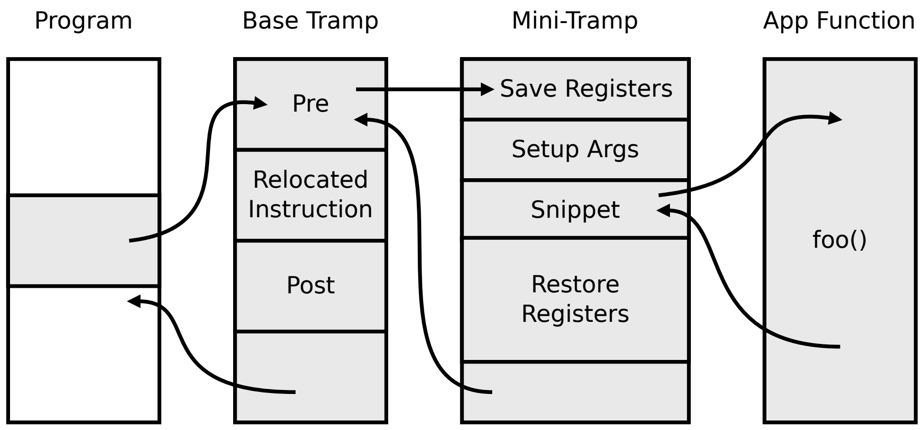

Figure 3.1: Trampoline patching mechanism [BH00]

corresponding function while the snippet would be the instrumentation code. Two further abstractions are *threads* and *images* typifying different lightweight execution threads or whole processes and static code images. The API also provides a sophisticated type system and a way to react on specific events realized by callback functions to the so-called mutator (program which is using the API to instrument the user application).

When it comes to the actual application of the patch, many different issues have to be considered. The OS services ptrace and procfs are used as utilized by debuggers or similar. Process execution can be observed, controlled, examined, and even changed in terms of core image and registers [ptr10].

A helper library is loaded as well including some utility functions and providing data arrays to store instrumentation code and variables. The snippets to be inserted are translated into machine code within the mutator process and copied into the data arrays (described above) in application address space. The most crucial point is then to carefully insert the branch to the patch code. Short code lines called trampolines are used to get the branching working and patch starting.

One or more original code lines are basically replaced by a jump to a base-trampoline to initiate the first branch. Then (after some preparation steps) immediately a jump to a mini-trampoline is done: registers are saved, some setup steps performed, the actual patch code is executed, and in the end registers are restored again. After coming back to the base-trampoline (from mini-trampoline) the initially replaced instruction is executed and some possible post procession done.

Figure 3.1 gives an overview of the patching process, the trampoline mechanism and the temporal alignment of the corresponding steps.

3.1.5 A Technique for Dynamic Updating of Java Software

[ORH02] deals with dynamic updating of Java applications in terms of exchanging classes at runtime without having to terminate the program execution. Classes may be added, deleted or even substituted dynamically. It is a strict software-based technique at program level guaranteeing several properties.

The principal procedure may be explained as follows: Classes which shall be updateable during runtime are pre-processed before deploying the application (swapping-enabled application). After this pre-treatment the hot-swapping mechanism can take place at runtime whenever a new class (for exchange) is available or any further action has to be taken. The following main properties are characterizing the overall system.

Semantics Preservation shall be given meaning that a swapping-enabled version of an application has the same functional behavior as the original one (from a black box point of view) and the updated version of an application has the same characteristic as if it was assembled like this from scratch.

Atomicity means that the update of more than one class is done in one atomic step to avoid any inconsistency problems between different parts of a program. When updating classes all instances have to be modified in parallel to ensure consistency and avoid different object versions (e.g. one original, one updated) at the same time.

Automation of Swapping-enabled Version is quite an important issue. The presented tool generates the updateable version automatically, almost without any developer intervention needed.

Runtime System Independence is emphasized at that point. No direct runtime support is needed, that is that any arbitrary compliant Java virtual machine can be used and proxy classes are deployed.

The technical effort of the different possible updating activities differs greatly, some are trivial. Adding is a relatively easy task as it is only required to add classes wherever they may be accessed via the class path. The new class itself is then loaded dynamically when an other amended class is calling the new one. Deleting a class respectively corresponding instances is also done in a virtually automatic way as the garbage collector takes care of non-referenced objects (if no class at all). Exchanging a class though is far from being trivial. At every location where the original class was created before the new one will be created in future. Furthermore all existing instances have to be migrated what may become a very sophisticated task. To make this possible and happen, it is necessary to pre-compute the respective classes (see above). The technical accomplishment of the updating process incorporates many steps at the preparation level (out of the original class a punch of helper classes and similar is created) and some special treatment is necessary for the actual application of the update. This is out of interest here.

The paper's results show the influence of the update application and subsequent execution on the overall performance in terms of time and memory. The original application and the swapping-enabled version were compared: in reference to time the overhead is very small (approximately 1% on average) while memory overhead is much bigger considered relatively (6% on average). The updating process itself requires time in the range of a few milliseconds. Moreover it is stated that the proposed updating mechanism was only tested in a very limited test environment (including a certain application, small set of subsequent versions, one dedicated test suite, etc.).

3.1.6 Dynamic Patching of Embedded Software

A very interesting approach referring to patching of embedded multitasking real-time systems is followed up in [ET07]. Everything centers on runtime patching for instrumentation purposes without preparing original source code or using dynamic linking. Instrumentation code shall be added into a running embedded system without ever changing source. Minimizing the influence on temporal behavior is one of the main goals. The process of applying a patch is automated, supporting inline coding in original source code and generating a patch out of the differences.

The patch process is defined as following: instrumentation, patch identification, target patch preparation and optimization, and target patch activation.

At the *Instrumentation Stage* code for instrumentation purpose is added manually or even automatically to the original source code. The user is in charge of taking care of the intrusiveness of the instrumentation code (mostly used for observations). There is by definition no direct code correlation between the patch and the outer original code. Usually a patch within this approach consists of `printf()` like statements and local or global helper variables. Anyway the idea could be extended to arbitrary patches.

At the *Patch Identification Level* the comparison between the patched and the unpatched object file is performed. Binary comparison is done on low level assembler but is yet configurable through the application of templates defining specific platforms. The process consists of two separated steps: disassembling the object header files (original and patched), and identifying changes in the object files regarding all specifics in terms of shifting code positions, etc. In the end an output file is created which next to the actual code includes adding and removal information.

Target Patch Preparation and Optimization is then the subsequent phase in the overall patching process. The output from the former stage is taken as input and optimization parameters are added. As result a list of specified changes for the target platform is generated. Patch commands are in the end supposed to be sent to the patch agent on the target. Optimization of the patching behavior like summarizing different patch positions can be done right here.

The last level is the *Target Patch Activation.* On the target platform a patch agent is already linked to the target system. Per patch a so-called trampoline is inserted which triggers the patch respectively the bypass to the new code. The instrumentation patch code is then executed respectively jumped at and after execution the return from patch is accomplished. If more than one patch is applied care has to be taken because the comparison with the original object file does not necessarily reflect the state on the target platform (after at least one patch was already deployed). So the patch agent has to keep track of the patch level.

In summary the actual patching procedure on target platform can be described as following: original assembler instructions (one or more) are overwritten by a jump to the trampoline at a specific desired position (done by patch agent task on target platform), the patch is executed, the overwritten assembler instructions are part of the patch and are executed, and finally a jump back is performed. When it comes to results and discussion it can be seen that the amount of trampolines needed and the patch location play a decisive role. Placing patches within loops means a huge overhead of jumping to the patch, jumping back, etc. and this leads to a code execution time raise up to 30 % and higher.

3.1.7 European and American Patents

Patching, especially on smart cards, is also a hot topic when it comes to patents in the European and the American legal practice. Many of them try to protect basic or ideas and approaches towards patching on specific platforms under certain circumstances by patents. In this short section some representative patents shall be shown which are closely related to the topic of the paper. Nonetheless it will become obvious that even this exclusive selection differs clearly from the approach under examination within the thesis.

The patent [Car02] is dealing with a so-called smart card patch manager. ROM based programs are to be patched on a smart card by means of placing code in a read-write memory which is then supposed to replace whole parts of original ROM code.

There is a ROM management record placed originally in ROM itself stating the locations and entry points for API functions and so on. Additionally there is a ROM management address indicator residing in read-write memory declaring where the ROM management record is located right now. The actual patch process comprises: placing new replacement code at an appropriate position in read-write memory, installing a new ROM management record (this time in read-write memory) and altering the ROM management address indicator in a way that it points to the new record. The new ROM management record then comprises the link information to the patched code which is activated and positioned in read-write memory ready for execution.

The patent [Hol01] is about a patching environment for modifying Java programs during execution on a Java virtual machine (JVM). The patch environment comprises a specific patch data structure (on computer memory) for modifying a loader environment of the JVM. The patented property is the patch environment that is able to handle a loader environment alteration online at runtime. The patch itself (containing the actual changes to the loader environment) shall be directly loaded into the patch environment and is executed at the appropriate point in time. Incremental changes are to be applied to a running Java application without newly reloading the entire program.

The loader environment which is defined as the logical memory area where information of loaded types or classes is stored needs to be patched herein. When a type has to be loaded due to the current instruction under processing a class loader takes the appropriate class file and passes prepared information to the JVM. Required values are stored in the loader environment (class variables, class method information etc.). The patch environment itself is already included in the specialized JVM environment (very similar to the loader environment above). A patch table data structure is present which is empty unless a patch is loaded through a patch file.

Two basic things can be conducted: altering an existing method body and adding a new method body by means of a patch. A patch file is generated after making direct changes to a target Java class file. The patch file is therefore sufficiently similar to generate a JVM patch structure in the special patch environment. The patching aware JVM itself then creates a new, patched-method table (which is exchanged with the original one), new patched-method body, etc. The patched-method table is in fact similar to the original one but it points to the new patched-method body. The patched-method table has then to be exchanged with the primary method table respectively activated by JVM. The actual application of the patch is synchronized in a way that the affected method is not under execution at application time. The old non-patched-method bodies are not deleted but retained in a way that a patch can be revoked after a while and the original state can be restored.

The patent [Gie08] submitted to the European patent office is about the execution of patches via application of caching mechanisms . An approach is presented and patented which describes the patching of code in ROM by means of patch code residing in EEPROM with the addition that patch code is loaded into a special cache for execution purposes while EEPROM programming takes place.

Patching initialization or triggering is done in a special way in this approach. There are distinct patching-exits defined in the original ROM code. The patching-exits are leading to any read or write memory location where the principal existence of any patch can be checked. Furthermore the applicability of a patch for the very exit point has to be verified. Relevant patch areas are then loaded to a cache and patch code can be fetched out of it while programming of the EEPROM (because of the common code instructions out of the patch) can be accomplished as usual. EEPROM writing and execution of the patch (originally located in EEPROM) can be done in parallel. Additionally, the code fetch out of cache is increasing the performance in comparison to EEPROM.

Many different cache strategies are used and contemplated in this patent's approach. A test run strategy is proposed where cache hits and misses are examined before the proper execution. There are several other cache strategies in use: One to be highlighted is the cache lock mechanism: the cache is preloaded via the test run and is then locked (respectively parts of it) for further updates. For a general short overview of the most common cache strategies see [Zho10].

3.1.8 Patching Mechanism and Enhancements in SCOSTA Smart Card Operating System

The master's thesis [Dat09] is about a special, selected patching mechanism in a SCOSTA (Smart Card OS for Transport Applications) implementation. SCOSTA [Nat02] is an open standard for smart card operating systems compliant to the closed ISO/IEC 7816 standards (introductive overview [Car10], ISO/IEC standards [ISO10]).

Three different basic approaches are proposed: modification of code image, function level redirection, and virtual memory address translation. Modification of code image means the direct alteration of code in place. That implies that the code is residing in alterable memory as EEPROM, Flash memory or similar. Firstly it is clearly emphasized that the relation between old and new patched code plays a crucial role and secondly that several entry points into the erroneous code (which is to be exchanged) may cause big trouble and the approach is thus doomed to fail. The main advantage is the simplicity and the unnecessary hardware support; only some more space may be needed in case the patch is bigger than the erroneous code segment. The biggest drawbacks are the missing flexibility in terms of entry points and the inherent disadvantage that code existing in ROM cannot be patched at all.

Function level redirection is another interesting way of patching as well. Therefore calling functions via dereferencing of function pointers is used in general to call any (patchable) function in code. A table of function pointers is maintained in an alterable memory and this table may be subject to changes in terms of redirecting any arbitrary function call. The new, patched function is then held in the alterable memory and is accessible whenever and wherever the old, original one was before. The big advantage is the inherent modularity of a function as a functional entity with defined input output behavior. The replacement mechanism is quite simple as well. The main disadvantage is that only entire functions can be patched what may become a real big overhead when thinking of a bug that is related to one single line of code only. Furthermore the corresponding compiler used in development has to come with the necessary features to support function pointers at all.

The one approach which is pursued and realized within the thesis is the virtual memory address translation approach. For this purpose the concept of virtual addresses has to be introduced where logical memory is seen as one coherent block (for storing arbitrary data) and is separated from the real physical memory for the purpose of extending physical memory or being able to map individually from logical view to physical location [Tec10]. Pages of virtual memory are directly mapped to physical pages by means of a page or address translation table. In the proposed solution a two-step translation table for mapping between logical and physical memory is used. One table resides in ROM, the other in alterable memory. When there is no patch present on the system the table in alterable memory (second level mapping table) is mapping one to one and has no actual influence. When there is erroneous behavior within one physical page the second level mapping table is altered in that way that it is pointing to the new patch code page instead of the original one. Thus loading a patch includes loading the corrected physical code page in alterable memory (into a special patch code area) and changing the second level mapping table to detour any accesses. The big advantage and main argument for this way of patching is clearly that patching of code existent in ROM is possible (no direct image replacement) and no special treatment (see function pointer approach) is needed. The obvious drawback is that only entire physical code pages may be exchanged. In the worst case that could mean that an error occurring in one single line of code would entail the loading and rerouting of a whole physical memory page. Depending on the page size the overhead may be huge. Figure 3.2 shows the basic idea of patching via alteration of the virtual address translation table.

3.1.9 Understanding Software Patching

In [Dad05] patching is considered from a more distant and process point of view. When we are dealing with patching we are very often going straight into technical details and overlook the settings and circumstances around the patching process. The meta patching process is examined and the software patching life cycle gets thoroughly described. The article's primary focus is on security related patches though the established practices can be applied to non-security issues as well. The overall patching life cycle process is described sequentially (from error identification to deployment) in the following paragraphs.

The very first life cycle stage is *Identifying The Problem.* As software is released, flaws and sometimes even quite obvious bugs can be found by various groups of people: the usual, dedicated user who is encountering a problem during arbitrary work, the hacker who is viciously or not (disambiguation, [Hac10]) attacking the software system or even staff from the vendor side that finds an undesired behavior after releasing a product. The ideal case is that the vendor or entity in charge is informed first, more precisely, before the (security) issue is made public. Then the software vendor is in demand to take action instead of possibly ignoring the problem.

Then the level of *Developing The Patch* is reached. The main goal is to release a patch

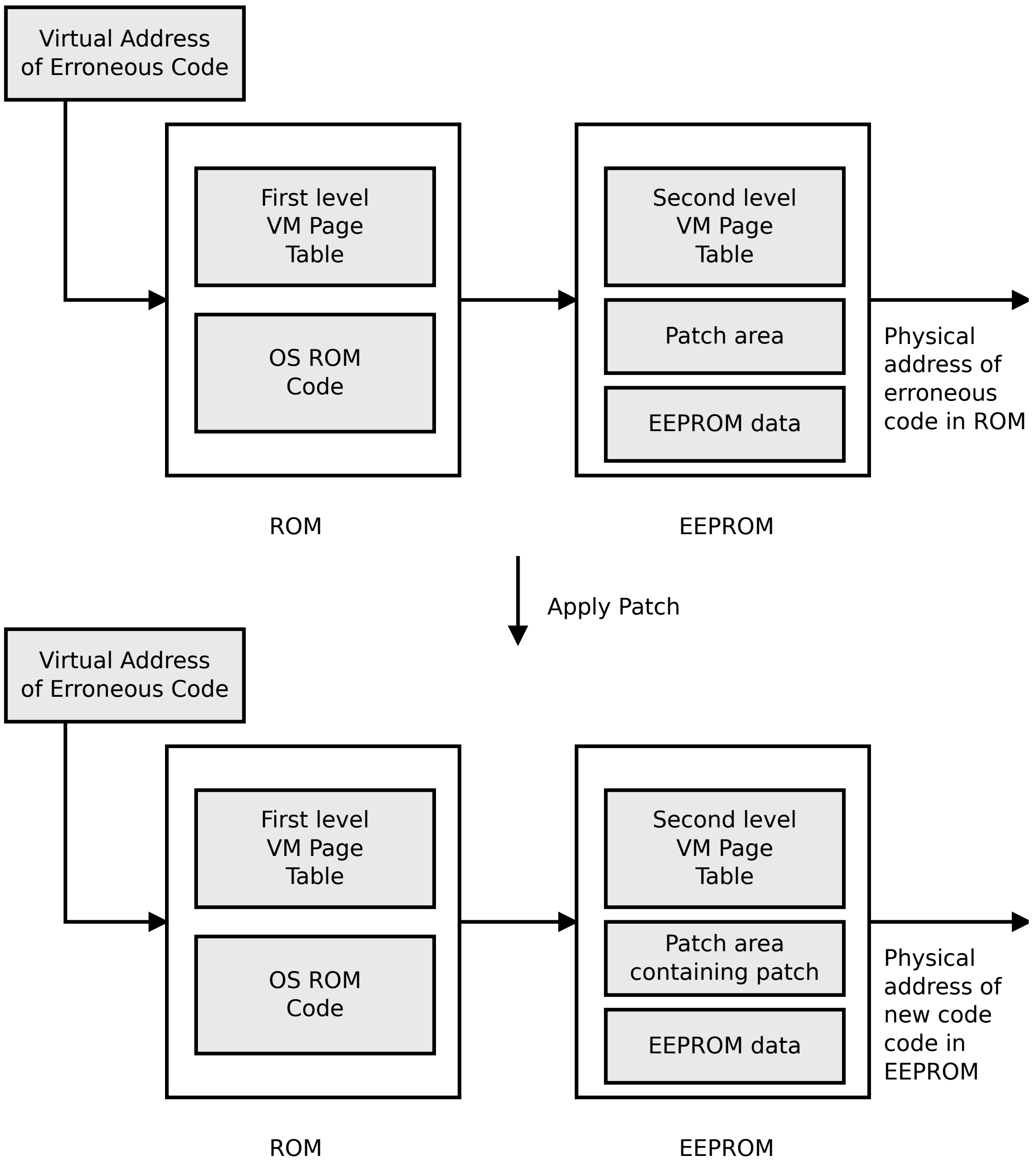
Virtual Address of Erroneous Code
First level VM Page Table
OS ROM Code
Second level VM Page Table
Patch area
EEPROM data
Physical address of erroneous code in ROM
ROM
EEPROM
Apply Patch
Virtual Address of Erroneous Code
First level VM Page Table
OS ROM Code
Second level VM Page Table
Patch area containing patch
EEPROM data
Physical address of new code code in EEPROM
ROM
EEPROM

Figure 3.2: Patching mechanism via virtual table address translation [Dat09]

without any negative influence on other functionality than the patched one (regression testing [AHKS93]) in a relatively short time depending on the impact of the flaw and the public awareness.

In the beginning the problem has to be understood thoroughly by the developers. While some bugs may be easy to fix some may have impact on many modules, may be architectural issues and have influence on many unforeseen things. All in all any side effects, similarities in other modules, multiple version occurrence (backporting), calling positions, etc. have to be investigated and therefore experienced developers are needed who have a deep knowledge of the module under consideration.

The design and creation of the patch is the main task after understanding the problem completely. Some quite big problems may slow down the patch development process: e.g. the module in question was not touched for ages and there is no owner for this or the developer of the module is not available anymore. When code is even in shared responsibilities and some entities are involved, the whole process becomes even more problematic and challenging. When developing a patch many compromises have to be made. The most important trade-off is between security and functionality and the schedule for releasing the patch. That is to say that one the one hand security shall be ensured on an agreed level and functionality shall not be limited due to the patch and on the other hand a very tight schedule is supposed to be met. The stability of the patch depends on the above-mentioned issues and appropriate decisions regarding the schedule-functionality trade-off. In the end of the development stage testing is a very important part. As it is also very time consuming to conduct testing thoroughly here the next trade-off is necessary. It may be bad for the patch stability to reduce testing to a minimum but it may delay the release when a complete test flow of an entire system is run through. Amending documentation (or additional patch documentation) and test plans, updating code analysis tools, etc. are final important steps which are crucial for a proper patch development framework.

For distributing a software patch a proper *Deployable Package* has to be built. Packaging formats are usually agreed on and have to be appropriately distributed. The user support process here depends heavily on the application area: widespread, fast distribution to many users or distribution to a selective user group. The current status of the device to be updated has to be considered as well. Patches may be associated and depending on the current version or patch level of a product different action has to be taken. The patch package structure can be tailored to that very fact of different preconditions of the system to be patched.

The distribution itself is not a side or easy task at all. Affected user groups (respectively affected products distributed to users), problem severity, fixing urgency, and potential mitigations for non-applying users are figured out. Patch side information has to be handled carefully as too much information (leaking out) accompanying the patch may support further attacks on unpatched products. The way of distributing a patch has to be secured carefully. That is on the one hand the availability of the distribution service (e.g., automatic distribution) and on the other hand the guarantee of the distributor identity (e.g., authentication via certificates is required).

Finally *Monitoring Status Of The Patch* is an important step in successfully applying all-encompassing patches. It is in principal the collection of feedback mechanisms for patch application. Several issues like successful patch installation, patched-software stability, patch distribution rate, etc. have to be considered. Some are technical issues (installation reliability, technical stability), some are project management related (customer acceptance and patch distribution).

3.2 Summary and Comparison

In the previous sections many different approaches and ideas can be seen. They all differ and view patching from different angles. In anticipation to the approach presented in this master's thesis it can be stated that all of them are very different, some more some less.

Especially [Meh10], [KLM09], [Ali91] deal with real-time issues and hot-patching for high-availability and highly responsive software systems. The focus is on the exchangeability of modules during runtime. Both the platform in principal (special purpose computers with rich resources) and the real-time requirements differ from the smart card patching problem. For a smart card which constitutes a strongly resource-limited device it is typically not necessary to patch during runtime.

[BH00] and [ORH02] show interesting approaches towards patching but are quite different from the idea of flexible patching on smart cards and are from other application domains such as simulation or similar. The most interesting difference to the presented approach of this work is that original executable code can be exchanged to apply a patch, as seen in [BH00]. Thus the flexibility is increasing while complexity of patch triggering and execution decreases. [ORH02] is using pre-treatment of original code to allow patching for specific modules. That is one fact that constrains patching in terms of correcting errors. Furthermore the approach is restricted to patch the loading of classes and is hence rather coarse grained.

[ET07] is dealing with the patching of embedded multitasking real-time systems and is coming a bit closer towards the smart card application domain. Yet there are many differences as again an in-place exchange approach is followed, similar to [BH00]. Trampolines are inserted to perform patches. The patching is not used primarily for error correction but for instrumentation purposes.

In Section 3.1.7 three patents towards patching are presented. They all differ strongly from the presented approach but some important commonalities can be seen. [Car02] is in the application domain of smart cards and provides a comprehensive mechanism of an API rerouting system. It is dealing with the typical environment of code in ROM and configuration possibility in alterable memory. The system is prudent but yet relatively coarse grained as only whole API functions can be exchanged. [Hol01] is dealing with smart function block exchange during runtime. Patch granularity is maybe the most important issue with regards to the presented approach. In this environment the virtual machine method table itself can be patched and reroutes to new and patched method bodies. Again in-place amendments are done and the idea rather applies to general purpose virtual machines. In [Gie08] a very interesting and smart card related approach is described. The most outstanding properties are the loading of patch into cache for fast execution and the existence of patching-exits where the principal existence of any patch can be checked. Hence the triggering strategy does not have much in common with the paper's approach and represents a kind of polling policy.

Eventually [Dat09] is dealing with a patching mechanism in the SCOSTA smart card operating system. The proposed solution introduces patching as the exchange of memory pages by means of virtual addressing. The virtual address table is altered and addresses then point to other physical locations. The patch granularity is high but the realization relatively simple in comparison to other patching mechanisms in the smart card domain. Furthermore minimal delay and good performance may be expected as there is no overhead for patch execution once the patch is placed. The theoretical paper [Dad05] is indeed interesting for every patching mechanism as it deals with many facts around patching. It can be understood as a guideline for a generic and comprehensive patching process even though it is particularly tailored to big software systems.

3.3 Novelties in the Presented Approach

In this chapter a lot of different approaches in the application area of patching could be seen. The approach presented in this master's thesis is following a similar direction to some extent but is yet different in some crucial points and introduces new ideas and concepts. The novelties are listed in the following:

- The presented approach enables patching of the Java layer of an SCOS stored in ROM.
- The presented approach enables fully flexible and fine-grained patching through application of EEPROM and other supportive hardware mechanisms.
- The presented approach enables high-performance patching through the applied platform mechanisms and full hardware support.

Chapter 4

Design of the Java Card Patching Mechanism

In this chapter the concept and design of the presented approach are described. Section 4.1 gives a concise problem description. Section 4.2 gives a short introduction and overview of the Java Card elements that might be patched and an analysis on probability of error occurrence and solving complexity. Next to the definition of the used Java Card patching terms in Section 4.3, Section 4.4 is presenting the requirements for the mechanism. In combination with the analysis this leads to Section 4.4.3 showing the Java Card elements to patch. Section 4.5 is giving the detailed design of the system. First the environment is defined, the patch code frame to embed patches gets detailed, various system views are showing the connections of the players in the overall mechanism, and the Java Card patch life cycle gets described thoroughly.

4.1 Problem Description

In this master's thesis a concept and mechanism to patch the Java layer of a smart card operating system stored in ROM is elaborated.

4.2 Patchable Elements

In a nutshell, the Java Card Technology establishes Java Technology on smart cards and other very memory limited devices (more in Chapter 2). On the one hand Java Card (understood as a language) is a real subset of Java [GJSB05] on the other hand a Java Card Virtual Machine (JCVM) similar to the Java Virtual Machine (JVM) is defined [LY99]. The limitations for the language subset are recorded in the Java Card Virtual Machine specification [Ora03] as well as the inherent requirements for the virtual machine running on the limited device. The specification is often referring to the original JVM specification and is following it in many ways. A very important issue for a virtual machine is memory management. The JVM specification introduces different runtime data areas which are subject to our investigations for the patching mechanism.

Before coming to the actual patching activity it is crucial to know what can be patched. The general overview is given in Section 4.2.1 and subsequently an analysis with respect to risk and complexity is done in Section 4.2.2.

4.2.1 Overview

In the following paragraph the basic categories of elements to patch shall be elaborated and a clear differentiation between the patchable elements is taken. Four different areas are defined in the following and are considered as patchable elements:

- Java Card executable bytecode
- Java Card stack
- Java Card heap
- Java Card method area

The first obvious patchable element is pure *executable Java Card bytecode*. Bytecode is an essential part of a Java class included in a method which is itself part of the class. Starting with the main method every piece of executable bytecode is contained in a method and thus the base for every Java application execution. Patching Java Card bytecode to change the principal application behavior is a potential option. The patching process may incorporate inserting, exchanging, and removing of arbitrary bytecode.

The second patchable Java Card platform element is the *Java Card stack*. The JCVM is a stack based virtual machine having a common stack as operand and call stack. That is, operands are stored interim on the stack for every calculation and method information (return address, parameters, local variables, etc.) is contained as well. Every operand on stack has a size of two byte. Figure 4.1 illustrates a Java Card virtual machine stack. The expressions for the different pointers to the specific stack areas are explained below. It may become necessary to add a local variable to the stack. Even removing or altering of variables in terms of their type is a possible choice.

A typical Java Card application is able to allocate memory dynamically whenever needed for storage of any arbitrary data. The Java operator `new` is the intended instruction for this. It allocates adequate memory space (depending on object type) on the *Java Card heap* for dynamic storage. In the JCVM memory is allocated in non-volatile memory. It is imaginable that it is required to patch any object on the heap in terms of altering values, structure, etc. Multiple instances of a class would have to be considered and patched likewise.

The fourth patchable element is the *Java Card method area* (including runtime constant pool, field and method data, etc.). The method area always stores per-class data and is needed whenever a class is loaded. The runtime constant pool, as a subordinate element, is created from class file constant pool and contains several kinds of constants (possibly numeric literals, references, etc.) and may hence be a target for patching. Fields (class variables) and method data (formal parameters, method modifiers) may be required to be altered in retrospect as well.

Theoretically it would also be possible to patch the *Java Card virtual machine registers* directly. These are the Java Card program counter (JPC), the Java Card stack pointer (JSP), the Java Card frame pointer (JFP), and the Java Card local variable pointer (JLP). The JPC points at the next bytecode that is going to be executed. The JSP is always

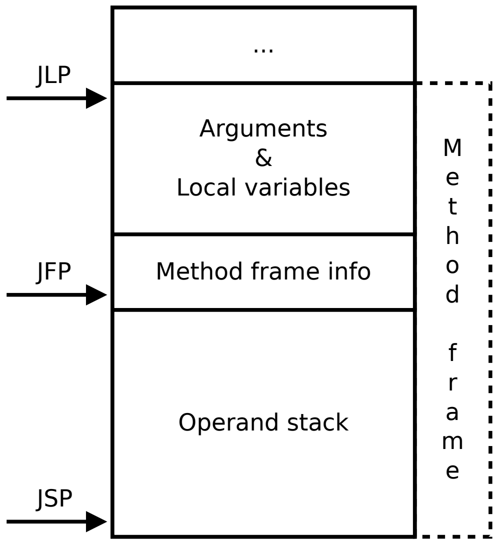

Figure 4.1: Java Card virtual machine stack

pointing at the top of the operand stack for the currently executed method. The JFP denotes the end of the current method frame info. The JLP is identifying the position of the first local variable or method argument. Direct patching of the noted registers is not considered in the course of this work. In Figure 4.2 a graphical overview of the different patchable elements is shown.

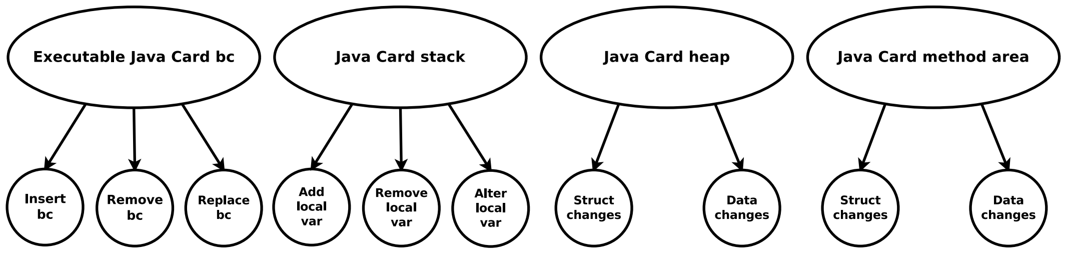

Figure 4.2: Patchable elements - bubble chart

4.2.2 Probability and Complexity Analysis

In Section 4.2.1 a comprehensive overview was given on what may be patched. In this section a first analysis regarding the patchable elements shall be performed. Two main questions are to be cleared up:

- How high is the probability that an error (to be patched) occurs in the respective element?
- How high is the complexity to fix an error in the respective element?

Table 4.1 shows a comparison between the different main patchable elements. For every element an empirical estimate is done in terms of probability of error occurrence in a final product and the complexity to fix the issue via the patching mechanism. A certain metric has to be applied: A factor for the probability of error occurrence is ranging from 0 to 10; 0 happens very seldom in practice, 10 happens quite often. Complexity is ranging from 0 to 10; 0 virtually no effort, 10 not practicable. The result is the reasonableness of developing a patching mechanism able to deal with the corresponding sub-domain. The value may range from 0 to 10 again; 0 very unreasonable, 10 very reasonable. The value is computed as follows (R...Reasonableness, C...Complexity, P...Probability):

$$R = f(P, C) = (P * (10 - C))/10 \tag{4.1}$$

Table 4.1: Probability of occurrence and complexity analysis

Patchable element	Prob. of occ.	Complexity	Reasonableness
Code	9	2	6.3
Stack	5	7	1.5
Heap	2	9	0.2
Method area	2	7	0.6

4.2.3 Analysis Details

Table 4.1 has to be understood as a very rough estimate though it shall be perceived as a guideline and the base for the design and implementation decisions in Section 4.4.3. The single patchable elements are considered in detail and the origin of the resulting analysis values are explained in the following.

Executable Java Card Code

The pure executable Java Card code is maybe the most important patchable element. The probability that a failure occurs right there and can be fixed by changing program execution is quite high and the feasibility in terms of fixing the problem is good. It is quite simple by definition to find the location of the executed code and be able to patch it. The complexity in detail though depends also on the impact the introduced code has on the out-of-patch area. There are code patches that do not influence the out-of-patch area at all. Others change the general memory demand (more in Chapter 5).

Java Card Stack

When it comes to patching the Java Card stack in terms of adding, removing, and altering stack elements, things become much more complicated but also less probable. The urgent necessity for another local variable on the stack is very seldom given and this applies even more to removing an element or altering its type.

Very often it is simply possible to circumvent the need for patching the stack by adding more code. The original need for local variables can be compensated by the usage of code patching mechanisms. The complexity concerning the manipulation of the Java Card stack is relatively high because many issues have to be considered (memory reallocation, stack referencing, increased complexity in code generation, etc.).

Java Card Heap

The probability of a failure residing somewhere in the Java Card heap or one which is only rectifiable right there by means of patching is extremely low. Usually elements on the heap are created via the `new` operator out of code while the initial information for an object to be created is located in the Java Card method area. Access to the objects takes place in the executable Java Card bytecode. Using the bytecodes reading and writing is possible. Patching heap elements as single dynamic Java Card objects is also very seldom desired as most of the time any existing instance of a class shall be patched. So it is much easier to patch the method area or the modifying code before object creation. The complexity for fixing such an issue may be extremely high. The point mentioned above is important once more: it may be hard to find all existing instances of a certain class or more generally spoken any other element on the heap. References have to be traced down to the very memory location. This would become necessary because on the one hand it is required when exchanging any value and on the other hand it is obligatory when it comes to do any structural change right there.

Java Card Method Area

Fixing the Java Card method area directly means modifying class fields, method definitions, the runtime constant pool, etc. It is quite seldom that failures have to be urgently fixed right there because errors or changes here are very often pointing at bigger structural changes which might become necessary, entailing a new ROM mask. Experience is teaching that structural faults on this level can be discovered in the pre-issuance development process. Hence fast fixing is not required and often postponed to later versions and new ROM masks.

An alternative may again be seen at code level namely at the very access code position of the respective field, constant or similar. Even though it is no real patching of the root cause the effect stays the same for single positions of access. Nonetheless, if an erroneous class field is accessed many times the patching on every code position where it is actually accessed becomes swiftly unfeasible.

One big issue towards high complexity and infeasibility to fix a certain bug is the fact that in Java Card constants (which are usually referenced via runtime constant pool) are already replaced by pure plain literals off-card in the CAP file conversion process. So the constants are not detectable on card anymore and there is no chance to fix a referenced Java Card constant globally as this information gets irreversibly lost.

4.3 Java Card Patching Terms

To get a deep understanding of the Java Card patching mechanism it is absolutely essential to get an idea of the underlying single elements building the system and a proper definition of the different entities within the overall design. Some underlying basic elements required for application in the patching design are detailed and explained thoroughly in Chapter 2. A comprehensive list of patching terms is given as follows:

- *SCOS configuration area* is an area in non-volatile alterable memory which allows to configure the SCOS in a flexible way implying behavioral changes.
- *Patching area* An area within the SCOS configuration area containing the patch image.
- *Java Card patching mechanism* defines the whole process of applying changes to the Java layer of the SCOS on the smart card. The overall mechanism ranges from Java Card patch generation to execution and possible deletion in the end. It takes place on and off card.
- *Java Card patching core mechanism* is the part of the Java Card patching mechanism taking place on card and is closely related to the SCOS and part of it, respectively.
- *Java Card executable* is the executable Java Card patch bytecode which shall be executed whenever a certain point in memory is reached.
- *Java Card executable environment* is the code area around the Java Card executable preparing bytecode execution.
- *Java Card patch* is defined as a Java Card executable and Java Card executable environment together.
- *Native executable* is executable native patch code which shall be executed whenever a certain point in memory is reached. Native patching exists next to Java Card Patching.
- *Native executable environment* is the area around the executable native patch code doing principal preparations.
- *Native patch* is defined as a native executable and native executable environment together.
- *Combined patch* is defined as compound of Java Card patches and native patches.
- *Combined patch environment* is the area around the combined patch preparing the system for any arbitrary patching activity. It is used in common for all applied patches.
- *Patch image* eventually is the whole patch (composed of combined patch and combined patch environment) and the one module which is loaded on the smart card.
- *Patch code frame* is the entire code skeleton and the totality of all environments for both native patches and Java Card patches which is ready for the patch code developer to insert the core patch code.
- *Address comparison module* is the module in charge of primary patch mechanism triggering on card. Initialization is done by means of the SCOS using values from the combined patch environment.

The terms patch and Java Card patch are used in parallel and in some cases the term *Java Card* is omitted to make the text better readable. The reader shall be mindful of the section titles which explain the context and thus make clear that patch is identical to Java Card patch. Within general sections the difference is clearly highlighted.

4.4 Requirements

In the following sections the requirements for the proposed Java Card patching mechanism are figured out. General customer requirements are identified to get an idea of what the topic is formally about and what is basically required. The system requirements are elaborated qualitatively on the one hand and with respect to quantitative performance characteristics on the other hand. System requirements are again divided into requirements for the patching core mechanism, the patch image placement, and requirements for the Java Card patch code generation.

Consequently Section 4.4.3 points out which patchable elements are considered worth to be patched in a comprehensive patching mechanism following both the analysis in Section 4.2.3 and the precedent requirements.

4.4.1 Customer Requirements

The main general requirements for the proposed patching mechanism are as follows:

CR1 Java Card bytecode as part of the applied SCOS or Java Card applet shall be patchable.

CR2 Java Card bytecode is residing in ROM and shall be patchable after masking in the fabrication process.

CR3 Java Card patch bytecode has to be deployable (in non-volatile alterable memory) and activatable after the masking process.

CR4 Java Card patch bytecode has to be erasable or deactivatable after original deployment.

CR5 The Java Card patch process (placement, deletion, etc.) requires proper authentication.

CR6 The Java Card patch process (placement, deletion, etc.) requires proper encryption.

CR7 Java Card patch processes shall be done in a (limited) secure environment.

CR8 Java Card patch processes shall be prohibited in some dedicated card states.

4.4.2 System Requirements

SR1 The overall patch deployment process shall be doable in fabrication or the pre-personalization phase outside fabrication.

SR2 The Java Card patch code development and deployment should take little time and cost in comparison to start a new ROM mask, given a flexible business case (choice to either patch or start new ROM mask).

SR3 The patch code frame size may not exceed `max_frame_size` bytes. The patch image size heavily depends on the patch code inserted into the patch code frame.

SR4 The patch core mechanism should be kept small and simple.

SR5 The switching overhead from original to Java Card patch code shall not exceed `max_switch_time` ns.

SR6 The Java Card patching core mechanism should have `loc_mem_size` bytes in RAM for the support of local variable patching.

SR7 The Java Card patching core mechanism must not require more than `loc_mem_size` bytes in RAM overall.

SR8 The Java Card patching core mechanism must not introduce performance issues and side effects to other code.

SR9 The Java Card patch code has to be checked for general security issues.

SR10 The Java Card patch code is required to be semantics-preserving. That means that functionality after applying a patch is the same as if the entire resulting code was written in this way from the beginning and patching never took place (alteration in original).

The particular values for `max_frame_size`, `max_switch_time`, and `loc_mem_size` have to be defined for every realization of the design.

4.4.3 Detailed Targets

In Section 4.2.2 a detailed analysis regarding the different patchable elements was presented. It can be stated that there are many different possible locations in a Java Card environment to apply patches to, dependent on the error to fix. But one crucial question is yet unanswered. What are the patchable elements that shall be subject to a comprehensive patching mechanism? What (realistically occurring) errors are going to be considered to become fixable within the presented approach?

Following the analysis, the most important patchable element is Java Card executable bytecode. It is most probable that an error occurs within any executable bytecode and it is necessary in a patching mechanism to allow the patching of executable bytecode as a basic requirement and primal step in establishing this mechanism. Patching executable bytecode is meant as the patching mechanism ability to insert, remove, or replace bytecode.

Applying patches on the Java Card stack is regarded to be relevant for inclusion in a comprehensive patching mechanism as well. It may occur that new local variables (used within a Java function) are necessary whenever new, extensive functionality is required. Removing local variables may be though not that important and rarely applicable as well as changing the type which is rather of theoretical interest.

Patching the Java Card heap directly is not considered to be helpful and really useful in any way. Analysis shows the difficulty in being able to patch accordingly and that the urgent necessity is nearly never given. The effort necessary to apply a patch on the heap exceeds clearly a reasonable degree (SR2). Patching the Java Card heap is not considered any further in the course of this thesis.

Method area patching is also not considered to be worth being included in a well balanced patching mechanism. Empirical observations show that many failures within this field can already be found in the development and testing phase before product issuance (more in Section 4.2.3). In comparison to the rare occurrence the complexity of fixing any failures is huge. Hence this patchable element is not subject to any further investigations.

A very interesting overall observation in the design process is that many failures may be (more or less) well fixable via code patching. A good example is the addition of local variables. Local variables are - observed from a distanced point of view - temporary storage for intermediate results of overall computations used for convenience and clear arrangement. Nonetheless these local variables may be omitted and computations are done inline. For most of the direct heap patching tasks it shall also be possible to accomplish the same goal through code patching. Especially the alteration of values on the heap can be conducted by addition of Java Card bytecode. Structural changes after primary creation though may lead to big troubles. There is no executable bytecode available for this (which could be added) and many further problems in memory management are coming up. Patching the method area is also partly possible by just adapting the executable Java Card bytecode (e.g., patching class field values by changing the value dynamically in the constructor or similar). Again, structural changes introduce many problems and no executable instructions are there for possible alterations. The case of patching constants is explained in Section 4.2.3.

When considering patching of different Java Card elements it should never be forgotten that the creation of a new ROM mask is always an option. The application can be changed on every position and all might be done from scratch again. However the applicability of doing a new mask heavily depends on the business case. Figure 4.3 points up the choice of patchable elements that shall be included in a comprehensive patching mechanism and are going to be under thorough investigation in terms of design and implementation.

4.5 Detailed Design

In the following sections the design of the presented Java Card patching mechanism is detailed. In Section 4.5.1 the overall patching environment for the patching mechanism is outlined. Section 4.5.2 goes deeper into the direct patching environment on card, the patch core mechanism. In Section 4.5.3 an overview of the patching mechanism is given showing the different components in static and dynamic collaboration. Section 4.5.4 describes the different stages in the patching process and the patch life cycle.

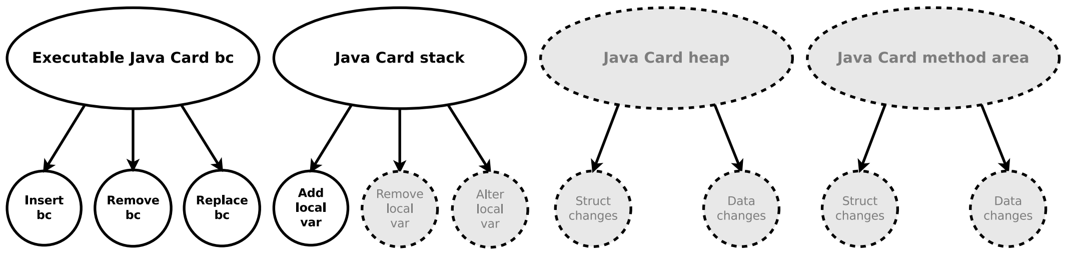

Figure 4.3: Choice of patchable elements to be included in a comprehensive patching mechanism - bubble chart

4.5.1 Patching Environment

This section shall be understood as a frame definition for the patching mechanism and a listing of preconditions to set a basis and to make the different design decisions clearly understandable.

An overall software system or SCOS is deployed on a small memory limited device or smart card in our approach. Within the SCOS a Java Card virtual machine is implemented executing Java Card applications which are inherently important for the proper device behavior. The Java Card bytecode is following functional and non-functional requirements but may be erroneous to a certain extent. The device itself contains non-alterable read-only, non-volatile alterable, and volatile alterable memory. The principal SCOS code is deployed on non-alterable read-only memory, some parts are located in the alterable memory. Mechanisms for writing to the alterable memory are in place. The device comes with appropriate communication, authentication, and encryption mechanisms to apply changes to the original software system in a secure way by placing Java Card patches plus environmental changes for patch initialization. For the placement the patch area within the SCOS configuration area is ready to use. The established overall system provides mechanisms to execute a patch flexibly. Possibilities to revoke a patch are in place. The patching mechanism per se allows to set both native and Java Card patches into the patch code frame resulting in a collective patch image. The image is the central entity considering application on card. For patch triggering the address comparison module is in use which gets its initial values via the SCOS which in turn is gathering the corresponding values from the patching area.

Considering patching environment the smart card is not the only supporting platform entity in use. For generating Java Card patches *developer devices* are necessary and are an integral part of the overall environment. *Transmission devices* are used to upload and apply patches whereas the nature of the corresponding devices and the process of patch placement may differ considerably. Finally the *interface operating devices* (e.g., card reader including a terminal and the user operating it) is the entity stimulating the smart card and is thus the root actuator for subsequent patch triggering and execution. Figure 4.4 explains the modular relations between different environment system entities in terms of Java Card patching.

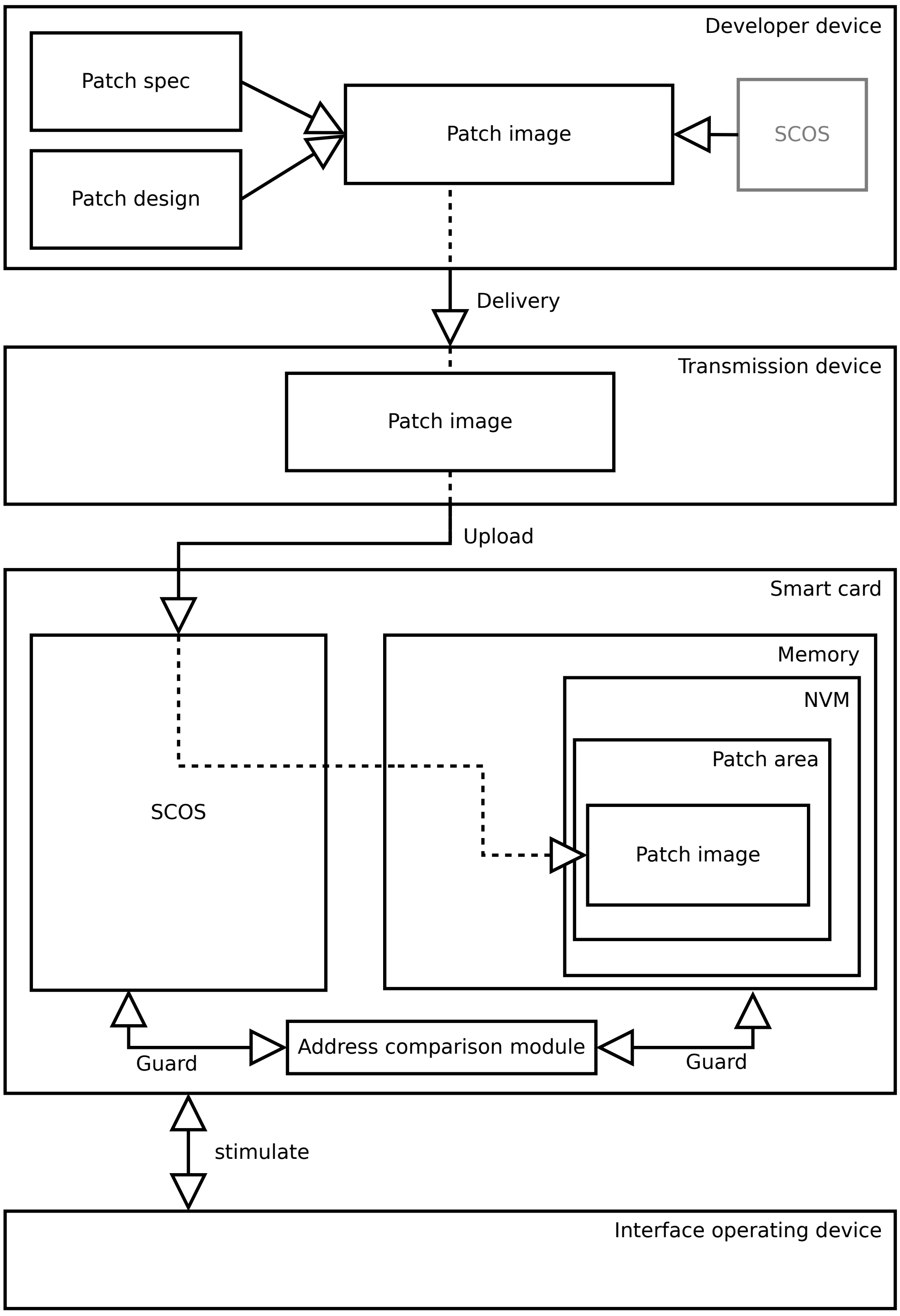

Figure 4.4: Java Card patching environment - overview

4.5.2 Java Card Patching Core Mechanism

To be able to establish an overall patching mechanism it is essential to have a surrounding mechanism in place on card. This mechanism consists of a patch code frame where the executable patch code can be inserted and a set of SCOS mechanisms and functionalities. The patch mechanism is able to serve a limited amount of native and Java Card patches. Some modular parts are shared amongst patches, some are particular for the single patch. The original platform (without any patches present) has to be prepared for the patching mechanism and thus parts are stored in ROM. The core part however is in non-volatile alterable memory and has to be flexibly changeable for the current prerequisites of all the contained patches. The single functionalities and dynamic behavior of the patching core mechanism are comprehensively explained in the following sections by detailing the patch life cycle in Section 4.5.4. Figure 4.5 shows the modular layout of a patch image containing all different submodules. The bare skeleton around the patch code is the patch code frame.

4.5.3 Overall System View

The design for a Java Card patching mechanism comprises many different modules and services. In this section the attempt to give an overall picture is done by showing various outline views.

The first view is a service-oriented modular view of the involved modules. Some closely incorporated services already exist and are reused, some are adapted for Java Card patching purposes, some are newly created from scratch (more in Section 5.2.1). The following services are specified:

- *Central patch management service* is comprising the entities collecting all and everything that is necessary for starting patch development, initializing the different processes and requesting miscellaneous services.
- *Basic patch image generation service* including all sub-services and sub-modules, is the very core module generating a patch image containing Java Card patches, native patches, and environments.
- *Cryptographic service* is a service handling keys, being able to authenticate diverse entities and perform encryption and decryption. This service exists both on-card and off-card. It already exists and can be reused.
- *Patch placement and deletion service* is a service in charge of placing and removing the patch image. This service exists both on-card and off-card. It already exists and can be reused.
- *Card connection service* connects the smart card with every outside entity and comprises all elements needed for communication on-card and off-card. This service already exists and can be reused.
- *Patch on-card service* abstracts all patching activities on the card as patch activation and patch execution.

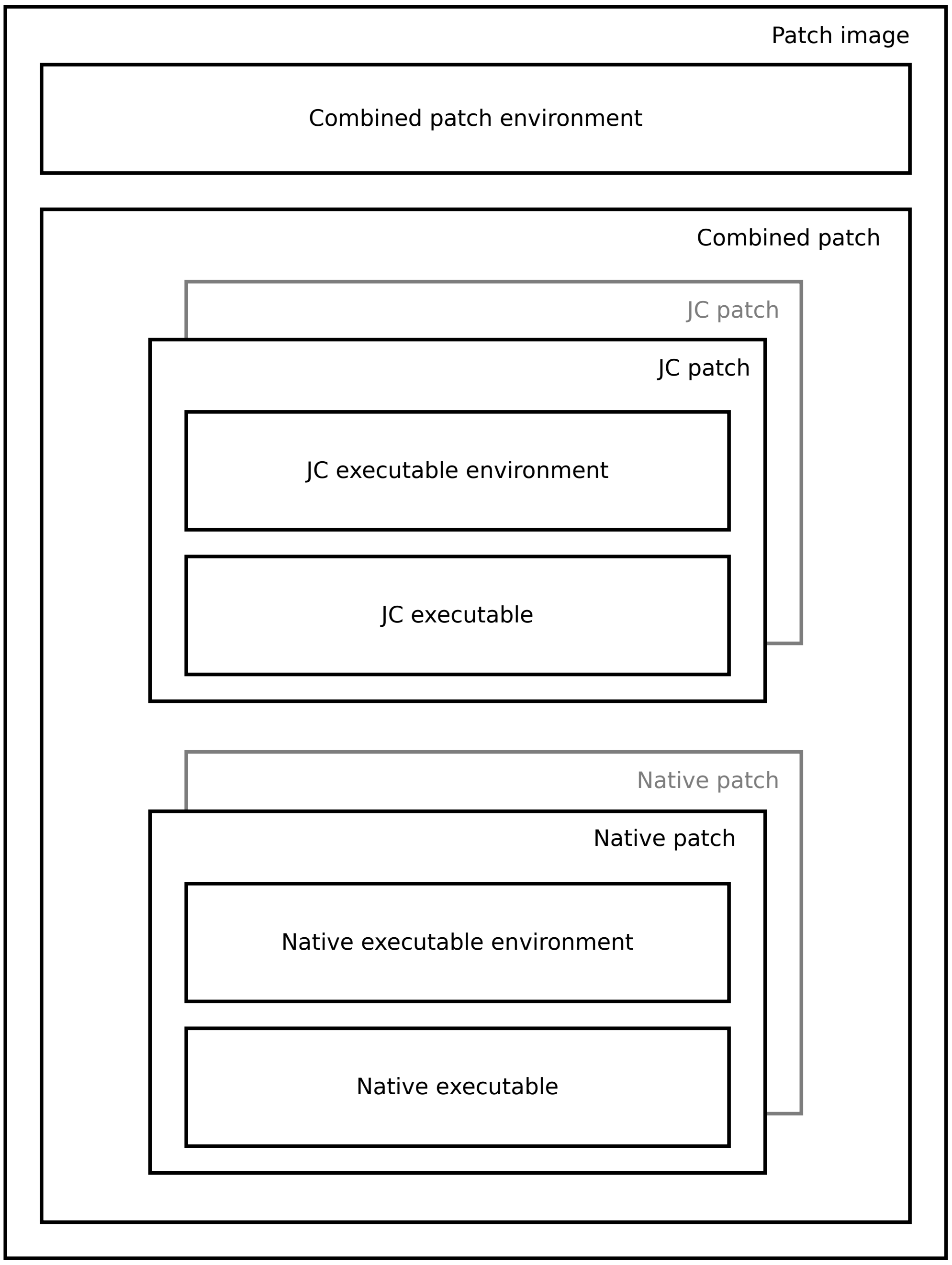

Figure 4.5: Patch image structure - overview

Figure 4.6 shows the statical logical relations between the different service modules (adapted logical view [Kru95]). The core service is the central patch management service using the basic patch image generation service and cryptographic service for producing a ready-to-use patch image. The basic patch image generation service is creating patches by Java Card or native generation and by the final usage of an embedding service. Java

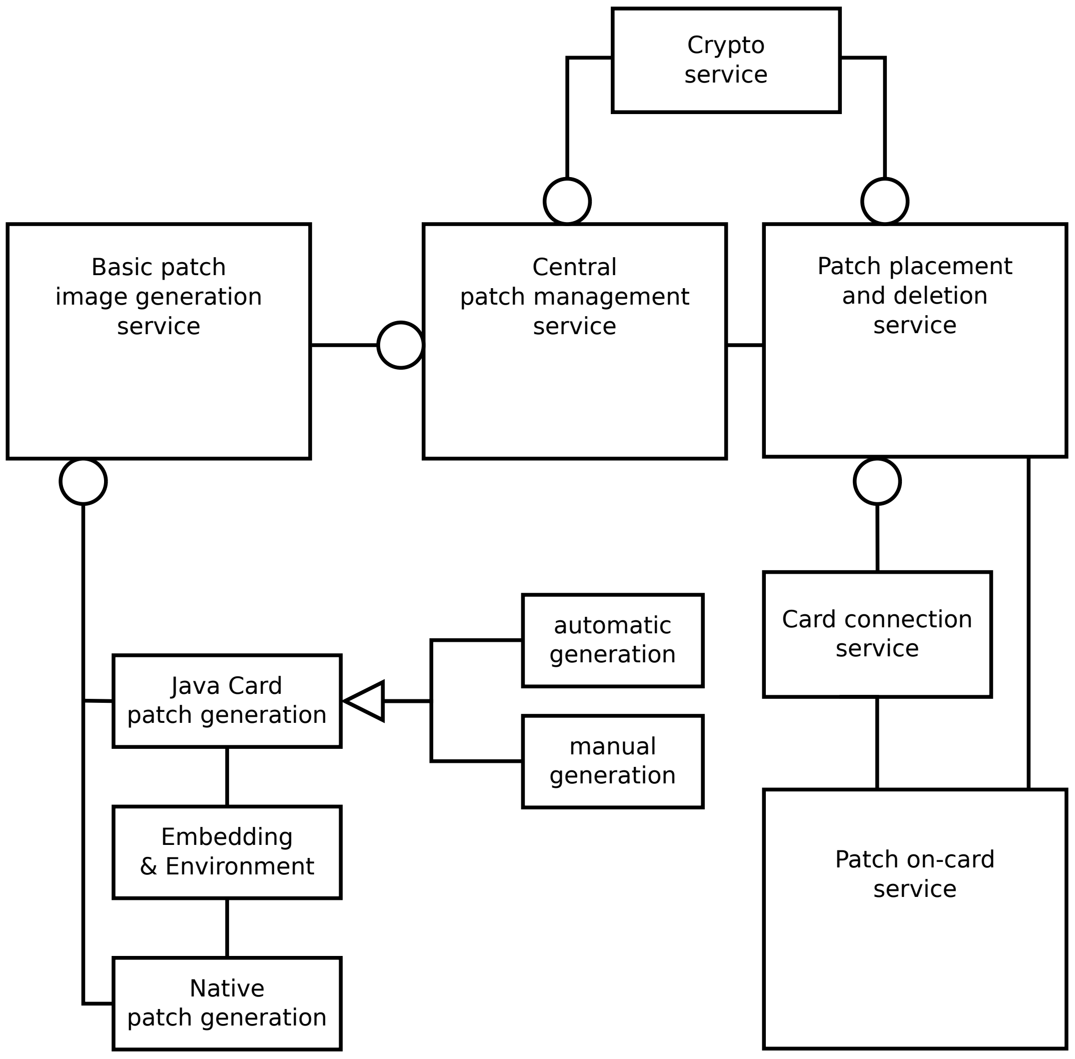

Figure 4.6: Logical view - overall patching mechanism

Card patch generation is either an automatic or a manual task. After generation the patch placement and deletion service is in charge of loading the patch image on card and removing it later on if required. Both is done by usage of card connection service (establishing communication link) and the cryptographic service. All patching activities in operative application as patch activation or patch execution on card are handled by the patch on-card service.

To get a dynamic view of the system a process model is developed. A typical software process view out of [Kru95] is adapted and raised up to the higher abstraction level of work flow processes. For the patching mechanism different processes can be identified and are defined as follows:

- *Patch management process* is the one central core process which is triggered by the occurrence of a patch request and an appropriate problem description. This process initiates all other processes, does the central management for patching and takes care of patch status and delivery meeting all security and procedural requirements.

- *Patch generation process* is the process comprising all steps for creating a patch image from core patch development to embedding and co.
- *Cryptographic management process* is the service-related process for delivering reliably secure keys, doing encryption, decryption and performing authentication. It is used several times in pre-delivery for patch completion and for the patch placement process.
- *Patch placement and deletion process* is the central process for placing the patch image and thus the Java Card patch on card. It is also in charge of deleting a patch. Various deletion strategies are thinkable.
- *Execution process* is the overall process for all patching sub-processes that take place on the card during execution: patch activation and patch execution

Figure 4.7 shows the patching overall system process view and emphasizes the miscellaneous relations of processes. The patch management process is central process which gets the patch request and initiates the patch generation process including all the sub-processes. The generation management process is the central point of control, and in charge of the allocation of tasks, testing, and delivery. It is decided whether to generate a native or a Java Card patch. After the generation the patch embedding process cares about the incorporation in the patch code frame and the patching mechanism. Finally the generation management process delivers the patch image to the patch management process. Potential encryption of a patch image is done by the cryptographic management process. In the patch placement and deletion process the patch image is put on the card and parts of the cryptographic management process have to be used for authentication and decryption. In the overall execution process patch activation and execution take place as separate processes. Initial patch triggering or stimulus is coming from outside simply by using the card in a dedicated way. The patch placement and deletion process finally has the task to remove the patch if necessary.

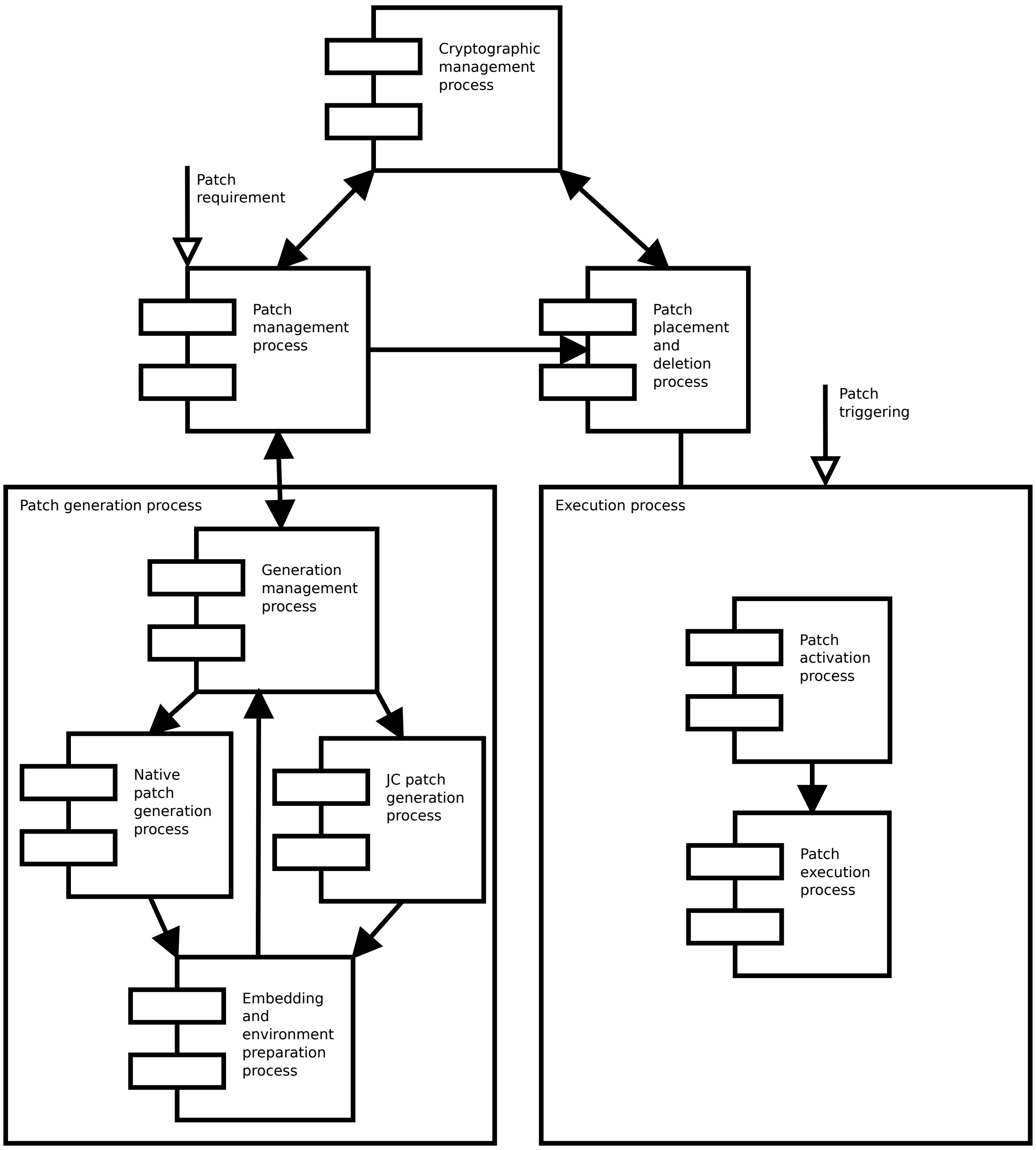
Cryptographic management process
Patch requirement
Patch management process
Patch placement and deletion process
Patch triggering
Patch generation process
Generation management process
Native patch generation process
JC patch generation process
Embedding and environment preparation process
Execution process
Patch activation process
Patch execution process

Figure 4.7: Process view - overall patching mechanism

Figure 4.8 represents a use case diagram of the patching overall system and highlights the embedding in the real world environment. In the following the main entities are described shortly:

- *Customer service system* is the platform for controlled customer and manufacturer communication.

- *Development system* comprises the entire development environment and all involved software and hardware development processes.
- *Placement system* is the patch placement environment and all involved organizational and technical processes for putting a patch on the card. Various different environments exist.
- *Application system* is the system in which the operative and testing application of the card take place. Various settings are possible, from application in the test laboratory to application in the field at the customer.
- *Reporting user* is the user in the form of a tester or customer.
- *Contact person* is the first person at manufacturer site who knows the problem.
- *Development team* is the entity working to fix the problem regarding all rules and constraints after having a comprehensive understanding of the problem.
- *Patch applier* is an entity outside or inside fabrication in charge of the placement of the patch on the card.
- *Problem reporting* is the initial use case comprising all the steps to communicate a problem in conjunction with the card.
- *Patch generation* is then the entire process to create an patch image for card application.
- *Patch delivery* is the hand-over of the patch image to the patch applier after ensuring full patch functionality within the overall system.
- *Patch placement* is the physical placement of the patch image on card.
- *Card delivery* is then the card hand-over after patch image application.
- *Card application* is finally the usage of the patched card in the field or in any testing environment.

Usually a reporting user who is a dedicated tester (in testing or certification laboratory) or any customer tells the contact person from manufacturer side about a problem with a specific type of smart card. After having clarified the exact problematic behavior and impacts on the work flow the contact person informs the development team about the issue. The development team has to understand the problem properly before starting the entire patch generation process. When the patch development is completed the patch image is delivered and applied by the patch applier outside or inside fabrication. After patch placement smart cards including patches are delivered to the reporting user and cards are under test or in operation.

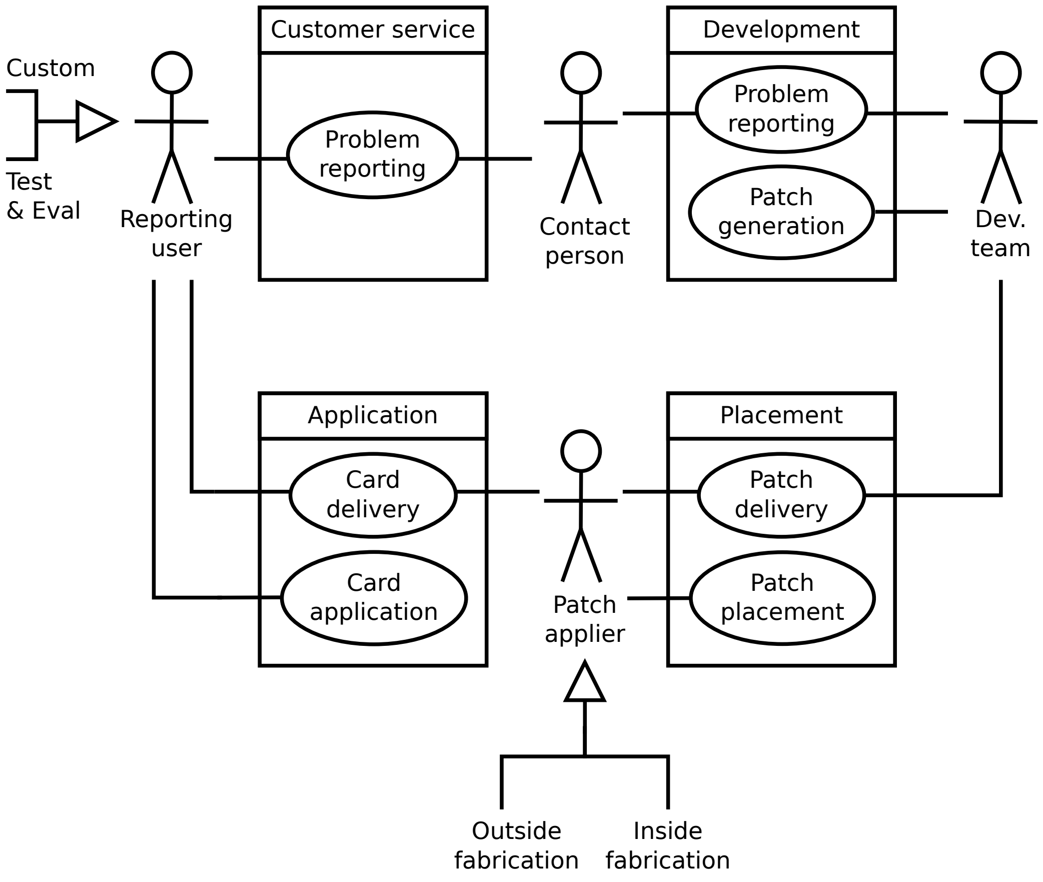

Figure 4.8: Use case diagram - overall patching mechanism

4.5.4 Java Card Patch Life Cycle

A Java Card patch runs through several main stages in the complete patch life cycle. The life cycle steps are considered to be sequential and are listed chronologically by the date of their occurrence. The understanding of the entire patching mechanism shall be given by the explanation of the different phases. The stages are as follows:

- Java Card patch generation
- Java Card patch placement
- Java Card patch activation
- Java Card patch execution
- Java Card patch deletion

It is left to state that between the different phases always a Java Card patch idle phase is slotted in where the patch is inactive and nothing crucial is happening in terms of patching. Figure 4.9 shows the life cycle of a Java Card patch including all different

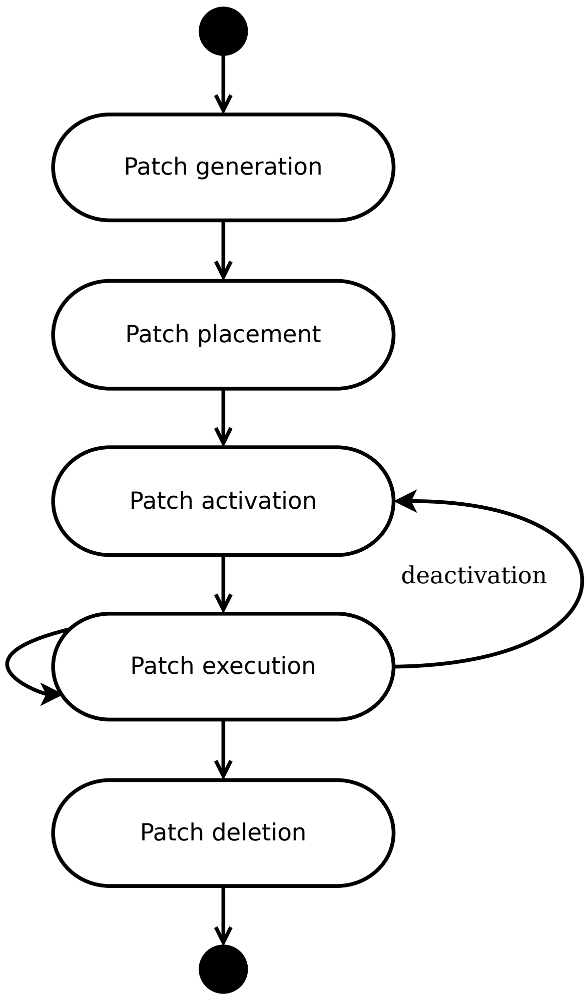

Figure 4.9: Patch life cycle

life cycle phases described in detail below. All typical transitions between the phases are highlighted and the time line runs vertically from generation down to a possible final deletion.

The patch mechanism is also characterized through the distribution to different physical entities. While patch creation takes place on developer devices and patch placement is acted out partly by transmission devices, the patching core mechanism is conducted on the smart card.

Figure 4.10 is showing the assignment of life cycle phases to physical entities or device types as it is implemented in the field. A more coarse grained division into overall stages is illustrated as well. Patch generation phase takes place on the developer device in the development overall stage whereas patch placement and potential patch deletion are conducted by transmission device and the smart card in collaboration. This is happening in the placement and deletion overall stage. This stage is usually entered twice as there is mostly a long period between placing and deleting a patch. Patch execution and patch

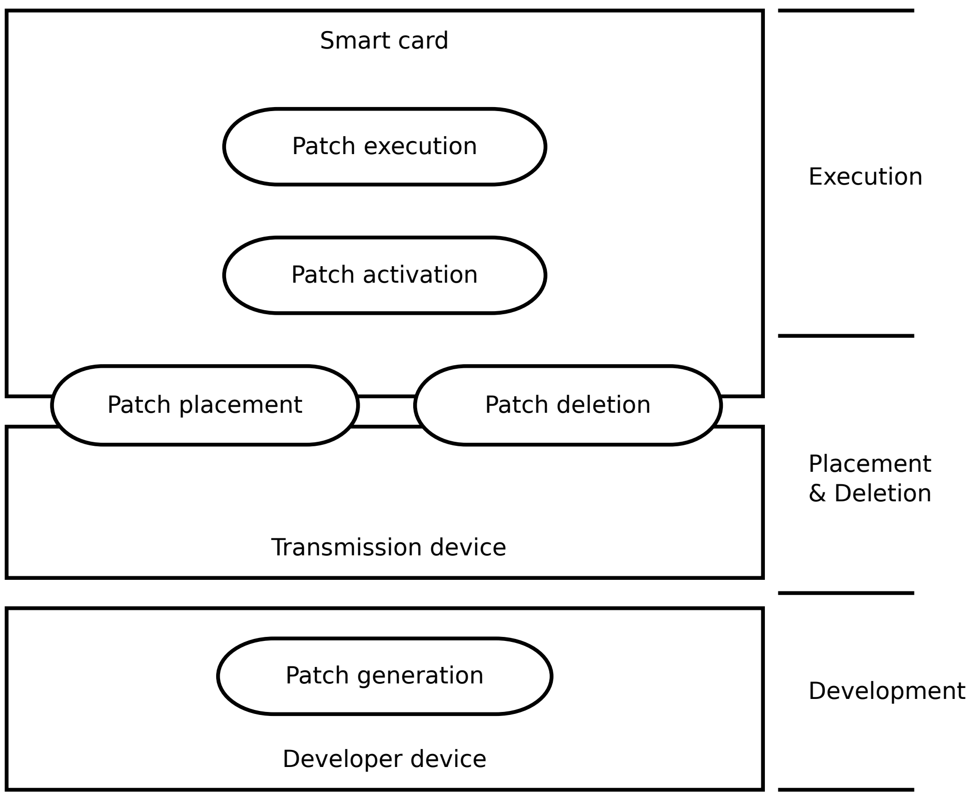

Figure 4.10: Physical division of patch life cycle phases

activation are proceeding on the smart card and belong to the execution overall stage.

Java Card Patch Generation

A patch life cycle starts with the generation of the patch and the appropriate preliminary steps. To state it more clearly, the generation phase can be further detailed.

An accurate problem description and a patch requirement specification are considered as the initial steps in the patch generation phase. As long as there is any lack of clarity, no action in the actual development has to be taken. The patch requirement specification is then considered as an outcome of the initial phase and recorded preferably in written form or verbally. Optionally a short concept, design, or detailed design document for the patch might become necessary depending on the depth and influence of the patch. This shall always be held in written form and specified clearly to avoid any ambiguities or misunderstandings. Security is already a big issue at this stage (SR9).

The actual patch generation for Java Card is again a multi-chain process. In principle it consists of generating a patch by appropriate means (more in Chapter 5) and the embedding into the patch code frame. The filled patch code frame is then referred to as the patch image (detailed definitions in Section 4.3). One patch image may contain several Java Card patches and native patches composite in the combined patch along with the combined patch environment. The term *patch level* is used to identify different patch

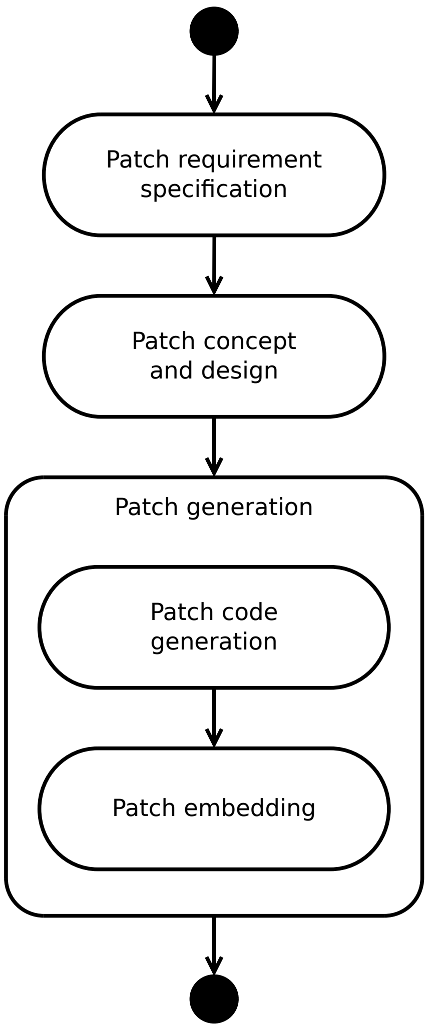

Figure 4.11: Patch sub life cycle - generation

images but does not make any statement about the number of included various patches.

Java Card patch generation is done in the original environment where any other Java code is written to ensure that requirement (SR10) can be met. The patch generation phase is maybe the most important and critical one as it is necessary to fulfill additional system requirements in terms of coding security (SR9) and development time (SR2). The quality of a patch is thus largely decided in a very early phase of the patch life cycle.

Java Card Patch Placement

The placement of a patch on the device is a crucial part of the entire patching mechanism. In fact the patch image is the entity which is placed on the smart card (including the Java Card patch). Patches are always loaded as a single unit. The placement itself always takes place in a (limited) secure environment. That means that patches are never applied in the field. Two general ways of applying a patch are defined (SR1). Appropriate security requirements have to be met (CR5), (CR6).

The patch placement in fabrication can be done without any encryption or authentication mechanisms as the environment is regarded as first level secure environment and protection is not necessary. It is done without any intervention from the card operating system side.

The patch placement outside fabrication is regarded to be a process in a (limited) secure environment. The patch image itself is property of the manufacturer and thus protected by means of encryption (CR5). Authentication, to have the ability to place a patch is required (CR6). Any interaction regarding patch placement is done via the card operating system which supports authentication and encryption appropriately. In certain card states patch placement is not achievable any more (CR8).

All mechanisms use non-volatile alterable memory where patch may be placed and activated, following requirement (CR3). In both cases fast placement techniques are used to ensure feasibility in the overall process. Yet the placement time heavily depends on the patch size and cannot be generally ascertained. General memory limitations have to be considered regardless of the different access mechanism.

Java Card Patch Activation

The patching mechanism is in need of a patch initiation and activation mechanism (CR3). It is ensured that a patch can be initiated after the device is in operative use (CR2). A flexible and easy-to-deploy activation mechanism is used in conjunction with the patch placement mechanism. The principal preparing mechanism that is leading to patch activation is already done in ROM and in place a priori, the core part residing in non-volatile alterable memory is explained as follows: The crucial step is to initialize a so-called address comparison module with respective values from the combined patch environment. The module becomes interesting in the patch execution phase as it is the principal triggering element. The patch activation steps are done in an early state of device operation in field so that the patching core mechanism is ready to use as soon as possible.

Java Card Patch Execution

Patch execution is the active and central phase in the patching life cycle. Basically, the phase starts with patch triggering, continues with patch execution initiation and patch code execution, and ends with the return to normal code execution. Figure 4.12 illustrates the different sub-phases.

Patch triggering is done in a flexible way so that access to any arbitrary memory position may trigger patch execution. The patch triggering sub-mechanism does not rely on certain points in code or locations in memory where patches can be triggered but allows it anywhere in memory. This can be accomplished via an exact address comparison mechanism whereupon the addresses are originating from the patch placement and initiation phase. Whenever reaching a point in code where the fetching address matches an address within the address comparison module the patch execution initiation is triggered.

The patch execution initiation phase is designed to prepare the actual patch execution and to bring everything in place, regarding time constraints (SR5). The execution environment may be prepared and set appropriately to allow displaced execution. Several issues and preparation steps have to be considered that do not really belong to the actual

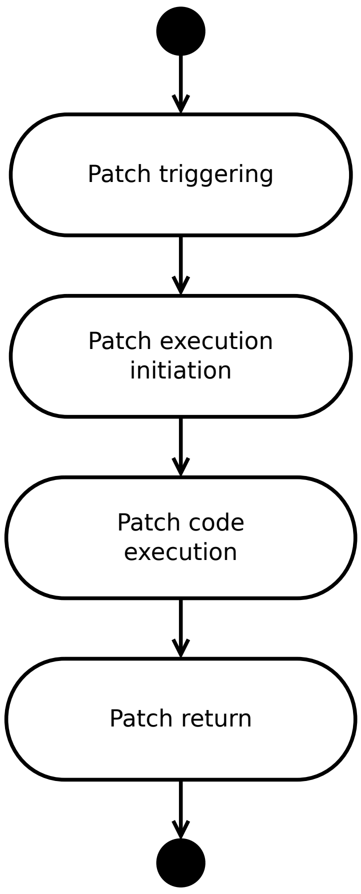

Figure 4.12: Patch sub life cycle - execution

executed patch code and are numbered among the patch execution initiation (details in Chapter 5). A patch always influences the original patched code to a certain extent and this fact is already considered within this life cycle phase.

The core phase is then the patch code execution phase. Code is fetched and processed from the patch area in place, no memory displacement is done. Patch code execution is done as if normal execution was accomplished. Security issues during execution are regarded, based on (SR9).

As a patch has a defined start and end the return from patch can be seen as a separate sub-phase. It is guaranteed that code execution can be continued the same way it was done before patch execution took place. To give a comprehensive example, from an outside point of view it shall not be noticeable that a function was patched within. Furthermore it shall not be noticeable for the after-patch code (inside the function) that a patch was executed right before. Generally no side effects are expected (SR8).

The overall patch execution phase itself may be repeated as often as required. From one execution phase to another different events might occur. Either there is an implicit idle phase where usual execution takes place between two patch execution phases or there is deactivation going on and a new execution requires a preceding activation. The incidence

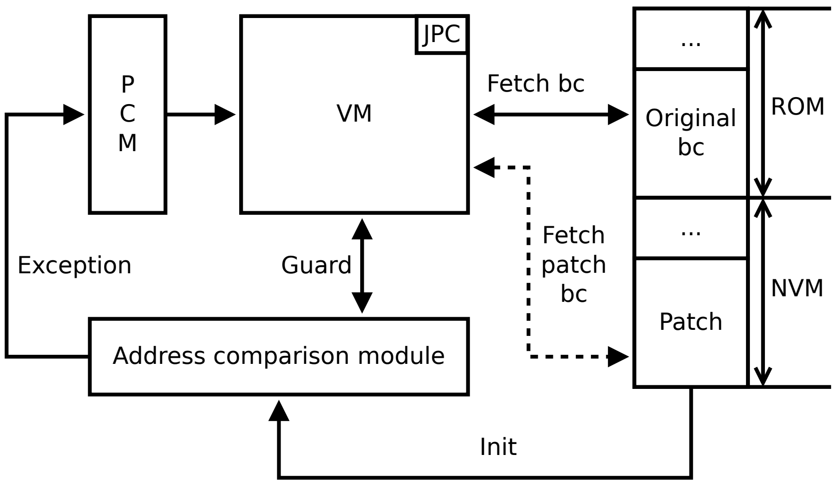

Figure 4.13: Patch triggering and execution - logical structure

of a patch execution process clearly depends on the work flow and the code position the patch is triggered from (qualitative evaluation in Section 6.2).

Figure 4.13 shows the logical structure for patch triggering and execution. The connection between the logical modules is highlighted. The VM executes bytecode out of ROM depending on the content of the JPC. When it reaches a certain position the address comparison module triggers an exception. The module is initialized with address data for patch triggering from patch area (combined patch environment) in non-volatile memory (NVM). Via the patching core mechanism (PCM) the VM is redirected to execute patch code (Java Card executable) out of NVM. The return from patch to usual execution in ROM takes place after the patch execution finished.

Java Card Patch Deletion

After having a patch applied and possibly executed it may become necessary to revoke it again (CR4). Generally, it is possible to delete a patch physically except dedicated card states are reached (CR8). The deletion is mostly achieved by a new patch image placement but can also be done by other erasing methods. Next to that it is possible to leave the patch image content on the card but render it functionally inaccessible by amendments in the SCOS configuration area.

Chapter 5

Implementation of the Java Card Patching Mechanism

In the following sections implementation issues are thoroughly considered giving a technical insight into the Java Card patching mechanism. It shall be shown how different modules are realized and how various detailed design decisions are implemented.

In Section 5.1 the general development environment is characterized. The common toolchain and specialties around the development are highlighted. In Section 5.2 a basic overview regarding the implementation of Java Card patching is given. This part shall be mainly understood as a view showing the technical relations between the different involved entities. Another part deals with the environment elements which are essential for the realization of the patching mechanism. Section 5.3 deals with the implemented Java Card patching core mechanism. Section 5.4 does a classification of the presented patching mechanism, based on Chapter 2. In Section 5.5 the different life cycle phases are technically detailed. Based on that, a technical description of the entire patching system is given. The patching phases are already known from Chapter 4 but the focus here is on the technical realization instead of a functional and abstracted description.

5.1 Development Environment

The development environment used for our implementation is very complex and sophisticated. The complexity of the work flow and the diversity of tools in use are residing in the nature of the underlying area of application and are historically grown as well.

The developer machines are usually equipped with Windows XP Service Pack 2 as general operating system. The build process for the SCOS is using typical Windows as well as Linux tools. The environment of choice is Cygwin 1.5.25 and the coherent Cygwin/X X11R7.5. The build process is crucial as it is utilized for many important steps within the patching mechanism development. A special build process for overall patch creation is in the field using Cygwin as well.

The applied compiler, assembler and linker landscape is very diverse as well. For the final target hardware and hardware related simulation a compiler-assembler-linker package from Keil in proceeding versions [Kei10] for the used 8051 derivate microcontroller [Aya04] is applied. For high level simulation the GNU Compiler Collection 3.4.4 [GCC10] and the

Microsoft Visual C++ tool collection 2003, MSVC 7.10 [Mic10a] are in use. The Java Card Development Kit version 2.2.2 [Ora10a] is used for applet development containing amongst others a standard Java Compiler and a reference Java CAP file converter. With regards to the CAP file converter for applets and the tool for the SCOS Java Card part a fully standard compliant proprietary Java Card converter is in use.

For coding tasks, different tools are used: for high level development on the Java layer or test applet development the Eclipse platform, Eclipse Europa v3.3 is used. For core development in Assembler or C the versatile editor Notepad++ v5.5 is in action. A possible alternative at least for C would be Microsoft Visual Studio. For code comparison the tool WinMerge 2.12 is in use.

From a hardware point of view a usual development personal computer is used, next to some special equipment characteristic for smart card development. Regarding testing environment it is referred to Section 6.1.

5.2 Realization Overview

In this section an overview of the realization of the Java Card patching mechanism is given. In Section 5.2.1 the work flow in the implementation is detailed. It is outlined what is done in the implementation and what parts are already existing as part of the underlying overall system. In Section 5.2.2 the environment of the patching mechanism is elaborated especially focusing on the elements needed for a performant and secure patching mechanism.

5.2.1 Thesis Work Flow

The presented Java Card mechanism is based on an already existing SCOS and a coherent patching mechanism for native code. As it can be seen in Section 5.2.2 some vital environmental elements are required for our implementation guaranteeing a qualitatively appealing solution. To facilitate this in principle it is necessary to have an appropriate platform which is providing the respective elements. A hardware related platform simulation is in place. The SCOS has to support these elements for usage in the Java Card patching core mechanism. It is amended and made available on the platform.

Another step is to create a Java Card patching core mechanism on card and to enable the platform to support Java Card patching. A patch code frame where to set in future Java Card patches has to be established. As the patch code frame is used for native and Java Card patching any side effects to native patching have to be considered. Furthermore the SCOS has to be amended to support the mechanism entirely as parts of the patching core mechanism are closely coupled to the SCOS or can be considered as a part of it.

Subsequently, to investigate and test the core mechanism, it is necessary to develop a Java Card patch generation mechanism. In the course of this task the mechanism itself is subject to further development and amendments.

Eventually a lot of effort is put into testing of the general mechanism behavior and possible situations in Java Card patching. Potential real world use cases were investigated as well. Results can be seen in Chapter 6.

5.2.2 Java Card Patching Environment

In this section the environment elements for the presented Java Card patching mechanism are elaborated. It is essential to understand the technical character of the single elements to get a comprehensive technical understanding. All the elements explained are used in the solution and occurring in the sections below.

General Elements

There are several general elements which are in principle necessary for smart card operation and are not purpose-built for the patching mechanism but closely related to some extend.

The SCOS seen as a functional entity has to be fully available on card for almost all steps in the patching mechanism. It is the central point of contact from the outside point of view (controls the smart card behavior) and the patching core mechanism is incorporated and part of it, respectively.

The Java Card virtual machine as part of the SCOS is very essential given that all is about Java Card patching. The central dispatching entity is the virtual machine, so at the end of the day both Java Card bytecode to be patched and the executable patch bytecode itself are executed on it.

Authentication Mechanism

Around the placement of a Java Card patch and the patch image, authentication is getting interesting and important.

For the patch applier outside fabrication authentication mechanisms are in place for usage in a (limited) secure environment. After verification and subsequent authentication of the outer entity by card the patch may be applied, following corresponding strict encryption rules (more in Section 5.2.2).

For the patch applier inside factory the option of patch application in the standard card production process without authentication and any intervention from SCOS is preferable. This is done within the EEPROM masking process.

Encryption Mechanism

Outside fabrication a patch is applied encrypted as the patch is property of the manufacturer and has to be protected accordingly. Appropriate encryption mechanisms are in use and decryption of the patch takes place on-card, virtually unrecognizable for the patch applier.

EEPROM

The central element for patching in terms of memory is the EEPROM module. EEPROM is the non-volatile alterable memory element chosen to be able to load any persistent information after the ROM masking process. This is essentially required for patching. A special area for patches is reserved in EEPROM (more in Section 5.5.2).

Watch Register

In the patch activation phase and the proximate execution phase the watch registers are very central elements. In principal the registers hold addresses that are compared against address values on the bus they are watching. Two basic families of watch registers exist: code watch registers and data watch registers. Code watch registers are monitoring the address bus for fetching instructions while data watch registers are observing the address bus for fetching operands. Whenever an address equivalence is discovered a hardware exception is thrown.

The following chapters concerning the life cycle phases deliver a deeper insight into the dynamic application of the watch registers in terms of initiation, update and triggering patch execution.

Hardware Exception

Hardware Exceptions or interrupts are fundamental concepts in computing. Whenever asynchronous, unexpected events occur an interrupt is triggered. Different types of interrupts can be distinguished by the system. For every type of interrupt a single interrupt handler or interrupt service routine is executed. The routine is located at a specific address, the so-called interrupt vector. In the presented approach this mechanism is very important as the watch registers are causing hardware exceptions ending in a specific interrupt service routine: the starting point for the patch execution.

5.3 Java Card Patching Core Mechanism

The Java Card patching core mechanism is the part of the Java Card patching mechanism taking place on card. It is closely related to the SCOS and part of it, respectively. The underlying SCOS infrastructure is already in place but has to be amended on several positions. As a native patching mechanism is in place the coherent patch code frame is extended and enabled for Java Card patching. Some other parts can be used together for native and Java Card patching, some are totally new and exclusive for Java Card patching.

As first implementation step the patch activation process has to be established. Therefore it is necessary to adapt the combined patch environment to set data watch registers appropriately. The triggering mechanism including exception handling and jump to patch area in EEPROM is largely the same as for native patching. However the combined patch environment and the patch code frame in general have to be extended to provide space for Java Card patch executables and being able to reroute to the specific locations in EEPROM. The VM has to be prepared for returning from initial patch triggering because after primary triggering a return to VM is necessary. A more precise description of the dynamic behavior is given by means of the patch life cycle phases in Section 5.5.3 and Section 5.5.4.

5.4 Java Card Patching Classification

As presented in Chapter 2 patches can be classified and divided into different orthogonal groups. Within this work Java Card patching is understood as patching of executable bytecode and virtual machine runtime area intended for execution on a Java Card virtual machine. The mechanism can be classified as cold-patching mechanism as there is no need and possibility to apply the patch during in-field operation. Real-time requirements are thus out of scope. In terms of patch size granularity it is potentially possible to patch every arbitrary size. There is no general limitation to function or module size. Regarding patch placement the presented solution is a real remote patching mechanism as it is physically not realizable to perform any in-place alterations to the present ROM code. The patching mechanism is hardware-supported and trigger-based by means of watch registers and hardware exceptions.

5.5 Java Card Patch Life Cycle

In Chapter 4 the functional characteristics of the patch life cycle phases are detailed. In the following sections the implementation details are worked out.

5.5.1 Java Card Patch Generation

Java Card patch generation is the first life cycle phase and a very crucial and complex one. Especially generating the patch code may become a real technical challenge. The embedding into the patch code frame for further processing is in contrast a straight forward task. The principal decision in the field of patch generation is automatic versus manual generation. While automatic generation means generation via compilation of source code manual generation is the writing of pure Java Card bytecode by hand.

In terms of patchable elements or patch target it is possible to patch executable bytecode and local variables in terms of adding variables. The following sections are handling the specialties of the single patching variants.

Executable Bytecode Patching

Referring to Section 4.4.3 pure patching of executable bytecode, without any addition of local variables or similar, is the main target. This is possibly the most important target by far. In fact the addition, removal or replacement of code is part of almost every Java Card overall patch.

It can be seen in the sections below that automatic code generation is mostly in use for executable bytecode patching followed by various manual amendments. Exactly these amendments are the biggest challenge when dealing with this patch target. So called special bytecodes are used which are substituting original Java Card bytecodes. Following special bytecodes are defined:

- Special bytecode for absolute addressing
- Special bytecode for maximum used stack size extension
- Special bytecode for returning from patch

There are some cases where bytecodes assume localities of functionality or use relative offsets as parameters. The offsets used in patch bytecode are naturally computed out of the off-card reality which is absolutely correct. The problem is that the patch is going to be applied on-card where the offsets and localities are not given anymore. This is especially true when patch code is referencing non-patch code (e.g., a function call). While patch code is located in EEPROM after patch placement the referenced code is most probably located in ROM. No locality is given and global referencing becomes necessary. For this purpose some special bytecodes using absolute addressing are introduced and are replacing bytecodes with a relative reference model.

Whenever a function is patched the basic environment for this function may be altered. This is mainly presented by the used local variables and especially the maximum used stack size which is required by the function. The need for an extension of the maximum stack size can be caused by new patch code. One reason may be the call of an additional function (more parameters on stack) or the bracketing for extra computations. To be prepared for such changes another special bytecode is amending the max stack size which entails in detail some more border checks, etc.

An executable bytecode patch is in need of a controlled return-from-patch. There are two basic ways for doing this. The first possibility is by means of a special bytecode including the return address in the bytecode parameter. The second way is to have a usual return statement in the patch or a Java exception exit that is reached in any case. The necessity for a special bytecode for patch return is thus depending on the general patch executable nature.

Local Variable Patching

As the second patchable element the Java Card stack in terms of adding stack elements was chosen. For the implementation that means that local variables are the entities of interest. Changing the Java Card stack for adding local variables is though a very sophisticated and tedious task. There are many reasons for that. One is that the core task of adding local variables to the stack introduces many problems. There is no standard bytecode which takes care of local variable creation in Java or Java Card. This is done statically at method invocation time and has to be repeated artificially - a special bytecode would be required to allow reallocation, displacement of parts of the operand stack, etc.

So the quest for a feasible solution is leading to a relatively pragmatic approach. A special purpose storage area for Java Card patching is placed at the disposal. Special bytecodes allow the storing and loading out of this area. This *patch storage area* is `loc_mem_size` bytes in RAM and divided into (two byte) short slots. One special bytecode followed by an index parameter (0 - `loc_mem_size` / 2 - 1) is used for storing elements, another one with a similar structure is used for reading elements. Any scope and reuse problematics have to be considered by the patch code developer. The collectivity of all Java Card patches has to be regarded as any patch could use the area for storing results and possible side effects are the consequence. In the end local variables are imitated without touching the original Java Card stack. The needed special bytecodes are as follows:

- Special bytecode for writing into patch storage area
- Special bytecode for reading from patch storage area

It is important to note that pure local variable patching does not appear by oneself but always in combination with executable bytecode patching. The writing and reading of the patch storage area is only reasonable when the emulated variable is used in code. Thus local variable patching appears as part of the in-code amendment phase in automatic generation as described below. A dummy standard variable would be used as placeholder in the generation process and afterwards replaced by the special bytecodes. Concerning realization on the platform it is referred to Section 6.2.3.

Automatic Patch Code Generation

The automatic patch code generation phase can be divided into several sub-phases: basic generation (including patch executable code generation, patch impact examination and possible patch width extension), in-code amendments, preparative pre-patching steps, and return-from-patch special treatment (last three are optional). Figure 5.1 is showing a phase overview of the automatic patch code generation (mandatory and optional sub-phases).

Automatic generation means the generation via build process by writing the desired target code and using the compilation result out of the process intermediate outcome. To be more accurate, first of all the desired code which shall erase the error is written in Java. The code is written right at the position where the old (erroneous) code was located to guarantee the most authentic location and behavior. In case of code addition the procedure stays the same: the generation is supposed to take place as close as possible to the original source code.

After first generation the produced Java Card bytecode is examined in terms of impact of the introduced source code. There are two different cases within this step: either the locality in terms of changes is given or the impact spreads out of the local changes. The former case is the simpler and more comfortable way as it is constituting a plug-and-play approach and it can be proceeded immediately to the next step. The latter case is much more complicated and often leads to a patch width extension. That means that the source code changes are introducing changes in bytecode that are not local but somewhere before or after the actual primary change location. That results in a patch that is much bigger and often has to be triggered before to ensure proper functionality.

A typical example is a possible correction in the `switch` statement in terms of adding a case label. Even though the changes may be somewhere in the middle of the `switch` statement, the changes in source code may entail changes in bytecode much more above, e.g. in the `switch` header doing the evaluation for the different case labels which might be even displaced after patching (even case label order is not always one by one). That also means that the whole `switch` statement has to be replaced just because of one structural modification - a typical example for patch width extension.

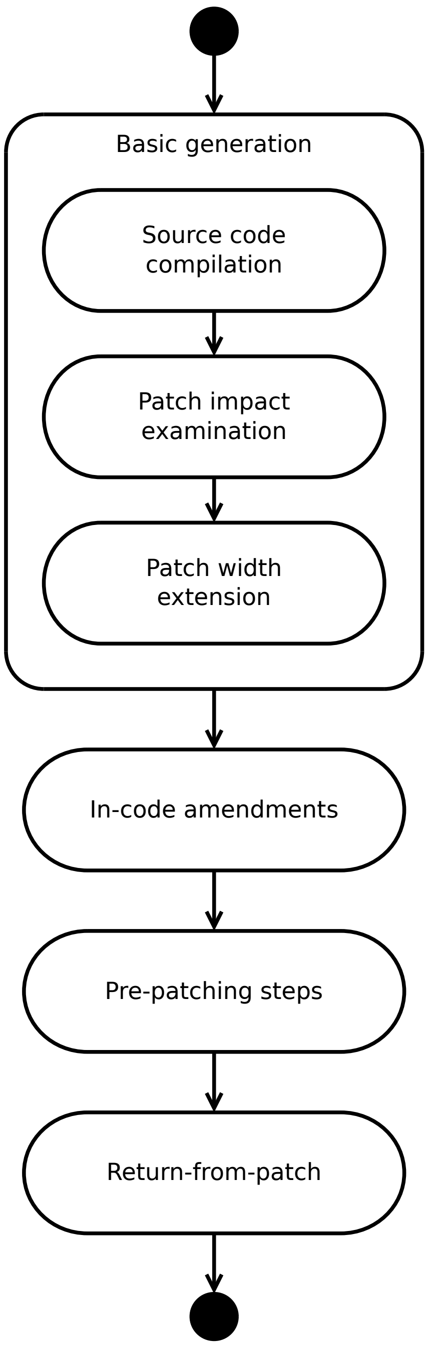

Figure 5.1: Automatic generation phases

Code Snippets 5.1 to 5.4 show the impact of source code changes on bytecode (in automatic code generation) using the example of a `switch` statement adding a new case label. It indicates the impact for the on-card code in terms of trigger location and return address. The original change is somewhere in the middle of the `switch` statement but there is impact on the `switch` header above. So the patch has to be triggered right at the `switch` header.

Listing 5.1: Original source code

```
switch{
  case A:
    functionalityA
  case B:
    functionalityB
  case C:
    functionalityC
}
```

Listing 5.2: Original bytecode

```
slookupswitch
  relref pos(A)
  relref pos(B)
  relref pos(C)

  ...
  pos(A): bc functionalityA
  ...
```

Listing 5.3: Altered source code

```
switch{
  case A:
    functionalityA
  case B:
    functionalityB
  case C:
    functionalityC
  —> Alteration begin
  case D:
    functionalityD
  —> Alteration end
}
```

Listing 5.4: Altered bytecode - patch width extension

```
—> Trigger address
slookupswitch
  relref pos(A)
  relref pos(B)
  relref pos(C)
  relref pos(D)

  ...
  pos(A): bc functionalityA
  ...
  pos(D): bc functionalityD
  ...
—> Return address
```

Referring to patch code simplicity and brevity it is also important to examine the created Java Card bytecode carefully. The situation is maybe explained the best way by means of an example. It is again the example of a `switch` statement. If the developer is adding source code within one of the case labels in the automatic patch code generation process maybe all other labels get dislocated. A direct comparison between the original bytecode and the patched one would maybe suggest that the patch has to be triggered at the beginning of the `switch` statement as the `switch` statement header changes. Nonetheless this is not the case and the patch can be reduced to the few bytecodes whose respective source code got actually changed. This is simply because the original structure of the `switch` statement remains unchanged on card and the patch code itself is just plug-in code and not changing the overall structure or any code.

Code Snippets 5.5 to 5.8 show the impact of source code changes on bytecode (in automatic code generation) using the example of a `switch` statement adding a few bytecodes. On card the impact is comparably low in terms of trigger location and return address as `slookupswitch` may stay unaltered. The labels are still valid and the new code is just plugged in.

Listing 5.5: Original source code

```
switch{
  case A:
    functionalityA
  case B:
    functionalityB
  case C:
    functionalityC
}
```

Listing 5.6: Original bytecode

```
slookupswitch
  relref pos(A)
  relref pos(B)
  relref pos(C)

  ...
  pos(A): bc functionalityA
  ...
```

Listing 5.7: Altered source code

```
switch{
  case A:
    functionalityA
  case B:
    functionalityB
    —> Alteration begin
    functionalityBplus
    —> Alteration end
  case C:
    functionalityC
}
```

Listing 5.8: Altered bytecode - no patch width extension

```
slookupswitch
  relref pos(A)
  relref pos(B)
  relref pos(C) //shifted

  ...
  pos(A): bc functionalityA
  ...
  pos(B): bc functionalityB
          —> Trigger address
          bc functionalityBplus
          —> Return address
  ...
  pos(C): bc functionalityC
  ...
```

In some special cases an additional phase comes into play after the primary code generation: in-code amendments. Depending on the patch target there are bytecodes exchanged by special bytecodes.

Preparative pre-patching steps might also become necessary and are incorporated in the form of further special bytecodes. Again, it is depending on the patch target. The basic environment for a function might change by a patch application and so dynamic amendments might become necessary.

Return-from-patch special treatment is another potentially required step. The necessity and realization is closely related to the patch target once more. The connective commonality is that the patch itself is aware of where to return to. For further information on manual amendments, see executable bytecode patching and local variable patching above.

Manual Patch Code Generation

In some very special cases it might become necessary to omit the step of automatic patch code generation through compilation. That means that the Java Card bytecode is written directly and manually from scratch. This approach is feasible for small changes only. The patch developer has to be fully aware of the virtual machine state at the particular patch time and may apply any allowed Java Card bytecodes.

In the presented approach pure manual patch code generation from scratch is not needed and utilized because for most cases automatic generation is applicable still including many manual steps. As seen above even in the automatic generation approach manual amendments and thorough investigations have to be done.

Patch Code Frame Embedding

No matter what way of code generation is chosen the patch code has to be embedded in the patch code frame. Technically this is a simple copy and paste of the produced Java Card bytecode (Java Card executable) and some amendments in the patch code frame. Referring to the patch model presented in Section 4.5.2 and Figure 4.5 there are amendments to be done in the combined patch environment. Actions have to be taken to enable a specific Java Card patch in principal and set the correct triggering address. This is the exact memory address of the bytecode where the patch shall be triggered. Usually the Java Card executable environment does not have to be amended. The amount of embedded patches is not arbitrary as the number of watch registers to activate is also limited.

5.5.2 Java Card Patch Placement

One very important step in the overall patching mechanism is the physical placement on card. There are two general ways of placing the patch depending on the physical card state, the environment security level and the entity initiating patch application.

One scenario is already taking place in a very young card state, the production at factory. It may become necessary to place a patch image after the masking process due to several technical or procedural reasons. EEPROM is written and masked in the factory and the patch can be placed in the special patching area within the SCOS configuration area. This is done all in one by means of a direct massive parallel writing process for EEPROM for a whole charge of smart cards.

The second scenario covers the patch application outside factory. It is obligatory to be authenticated to the card to be able to place a patch. The patch code is furthermore encrypted because the original patch code has to be protected as property of the manufacturer. Technically seen, all activities are conducted via application protocol data units (APDUs) and card acceptance devices on the host side. After successful authentication the patch data has to be appropriately treated and split up to fit into APDUs to be sent to the card. It is then placed step by step in the correct patching area within the SCOS configuration area. The reassembling and decryption is done on-card. Correct placement in the patching area within the SCOS configuration area is the desired result. Independent of the particular scenario in a card life cycle, the result of the patch placement shall be the very same. The completeness of the patch placement process is the precondition for a fully functional card afterwards.

Patches are never applied in the field or during operation because of security reasons. That means that the environment of patch application and the card state meets adequately high security levels. At some defined card states there is no more patching possible.

After placement every patch image is identifiable by a coherent patch level which can be requested by any host application. It co-determines the smart card identity in terms of SCOS behavior. There are never any incremental patch updates, one patch image always overwrites another.

5.5.3 Java Card Patch Activation

After the placement of the patch image including Java Card patches it is necessary to perform an appropriate activation for every included patch (executable patch code and environment for Java Card or native code). As already stated above in Section 5.2.2 there are watch registers on the platform monitoring the buses for instruction and operand addresses. As these registers are volatile memory they have to be initialized after every power-up. The task is to set the target addresses for the patch triggering and execution into the available watch registers.

The way of doing this has to be flexible and adjustable as the initialization only has to take place on a patched card whilst the ROM code is equivalent for the same card in unpatched state. Via the SCOS configuration area a flexible behavior can be achieved and the setting of the single watch registers is a part of the combined patch environment. In case of a Java Card patch a data watch register is set because Java Card is always fetched as data from a hardware machine point of view. The data watch register gets simply initialized with the logical address of a bytecode in ROM. When fetching at this very address Section 5.5.4 gets interesting.

The watch register setup shall happen as fast as possible in the smart card runtime (between power-on and power-off) because if there was an error in code before the initial activation sequence this one would not be patchable.

5.5.4 Java Card Patch Execution

After correct patch activation took place the next life cycle phase may come to its execution. It consists of several sub-phases: from initial triggering, to preparation and execution to the point of return from patch.

For a Java Card patch everything starts with the fetch from a bytecode address which is referenced in a data watch register. The Java Card virtual machine exists in memory and is written in usual native code that is fetched by the processor via the code bus. The basic bytecode fetch itself happens directly in the core of the virtual machine and is a pure data fetch for the hardware machine. Right when fetching from the corresponding address listed in the data watch register a hardware exception is triggered and an exception or interrupt handle routine is executed, respectively (more in Section 5.2.2). Figure 5.2 illustrates the basic hardware structure on the platform hardware.

The basic routine leads directly to the program patch environment where some preparation steps are done: care has to be taken of the platform (exception) state, it has to be figured out which watch register really caused the exception, and some special steps have to be performed that prepare the card for the patching mechanism and the return to usual execution. This area is shared between Java Card patches and native patches. The forwarding to the correct patch then leads directly to the patch code execution phase and a minimal set of Java Card typical preparations.

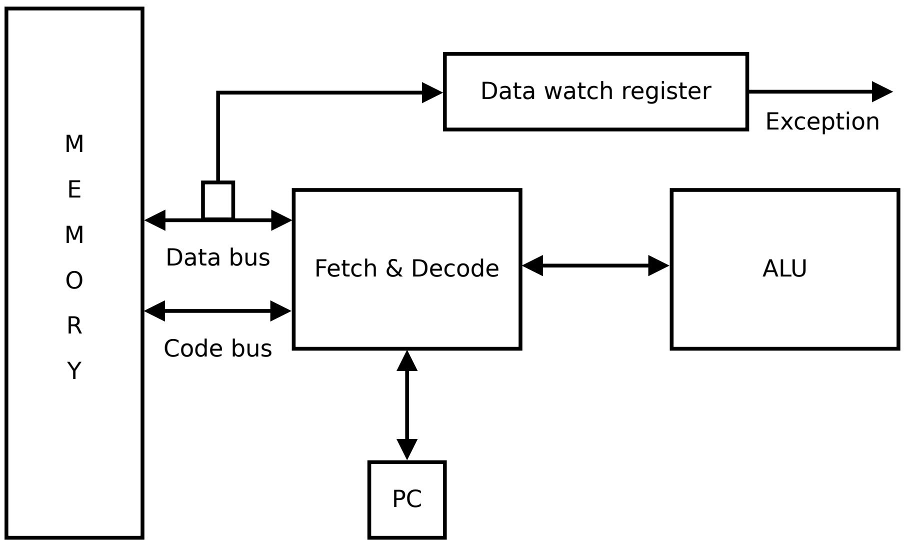

Figure 5.2: Patch triggering - hardware structure

After being directed to the correct position and thus the right Java Card patch, the steps are straight forward. Set the Java Card program counter to the Java Card patch code and return immediately to an appropriate point in the virtual machine. In the following the instructions out of the Java Card patch area are executed (Java Card executable).

The return from Java Card patch can be done in the two different ways as described above. Either direct return via the Java Card executable code (return statement, etc. in patch code) or return via special bytecode. This is the last touch with the Java Card patch area within one execution phase. After return from patch common bytecode execution can be continued.

As we have seen in Section 5.5.3 a Java Card patch has to be activated correctly. Deactivation indeed may occur because of several various activities. A Java Card patch is considered as activated whenever the reaching of the patch triggering address leads to patch execution. It is considered as deactivated when the behavior is not like this. One possibility of deactivation is the simple implicit deactivation through power-off. The watch registers are volatile memory and loose their states when power is cut off. So after every implicit deactivation an activation has to take place again. This usually happens in the activation phase at startup as described above. Another possibility is that deactivation takes place directly and targeted mostly by means of another patch. A reason for this may be the reuse of a watch register for two different (Java Card) patches. It may happen that a patch appears only once in a smart card runtime. If exactly this occurrence profile can be guaranteed the patch is allowed to be deactivated and another patch may use the same watch register within one smart card run.

5.5.5 Java Card Patch Deletion

A Java Card patch deletion marks the end of a patch life. In the process where a new patch image is placed in the SCOS configuration area the interrelations are as follows: In the beginning the old patch image was placed and a specific Java Card patch got activated every time after power-on in the SCOS startup. The old patch image is then replaced by a new one (not containing the specific Java Card patch anymore). By means of a reset which is conducted after patch image (re-)placement implicit deactivation takes place and the watch register is not filled with an address value for the original Java Card patch anymore.

Physical Java Card patch deletion is in principal mostly a replacement process of one patch image by another changing the patch level. Very often a dedicated Java Card patch is overwritten by an improved version or is staying binary the same (can be seen as deletion and new placement). The overall patch image may also get changed on some other position because a different patch was added to the patch code frame.

Chapter 6

Experimental Results and Discussion

In this chapter the results of the practical work are presented, discussed and evaluated. Section 6.1 gives a short overview of the testing environment in use. In Section 6.2 the principal approach is examined, especially in terms of its strengths and weaknesses. Characteristics and difficulties are detailed and the different issues concerning implementations are highlighted. Section 6.3 is showing measurement results concerning the basic patch mechanism. Subsequently a typical use case for Java Card patching is presented in Section 6.4. The interested reader should get an impression how a patch may look like with the help of a practical example. Section 6.5 shows a practical example of the impact of patching on project costs.

6.1 Testing Environment

For hardware related simulation and emulation on testing hardware an IDE from Keil, called uVision v4, is used allowing, accurate debugging. For communication to the hardware related simulation in the test environment a proprietary tool called SCComm2 v1.0 is applied. A powerful alternative would be to use a script shell called JCShell but this tool is not available for the applied hardware related simulation yet. For high level simulation Eclipse Europa v3.3 (Java and C) or Microsoft Visual Studio 2003 (C) are employed using the existing debuggers and having special plug-ins in place. As it will be seen in the following the hardware related simulation is restricted to some extent. The disregarding of EEPROM access times in the simulation may have the biggest impact on the results for the presented patching mechanism. Theoretical considerations concerning that topic are done in Section 6.2.4.

In the presented approach hardware related simulation is done only, no emulation on testing hardware so far. For future work on the topic typical hardware equipment will be in use. Different kind of smart card readers (contact and contactless) are at hand: Omnikey [HID10], SCM [Mic10b] and the Pegoda reader [Sem10]. For chip emulation without debugging and hardware debugging an emulator box from Ashling and the appropriate PKSC v1.5 upload and configuration tool [Ash10] is operative. In all test scenarios the build under test comes out of the original build process. To take the EEPROM access

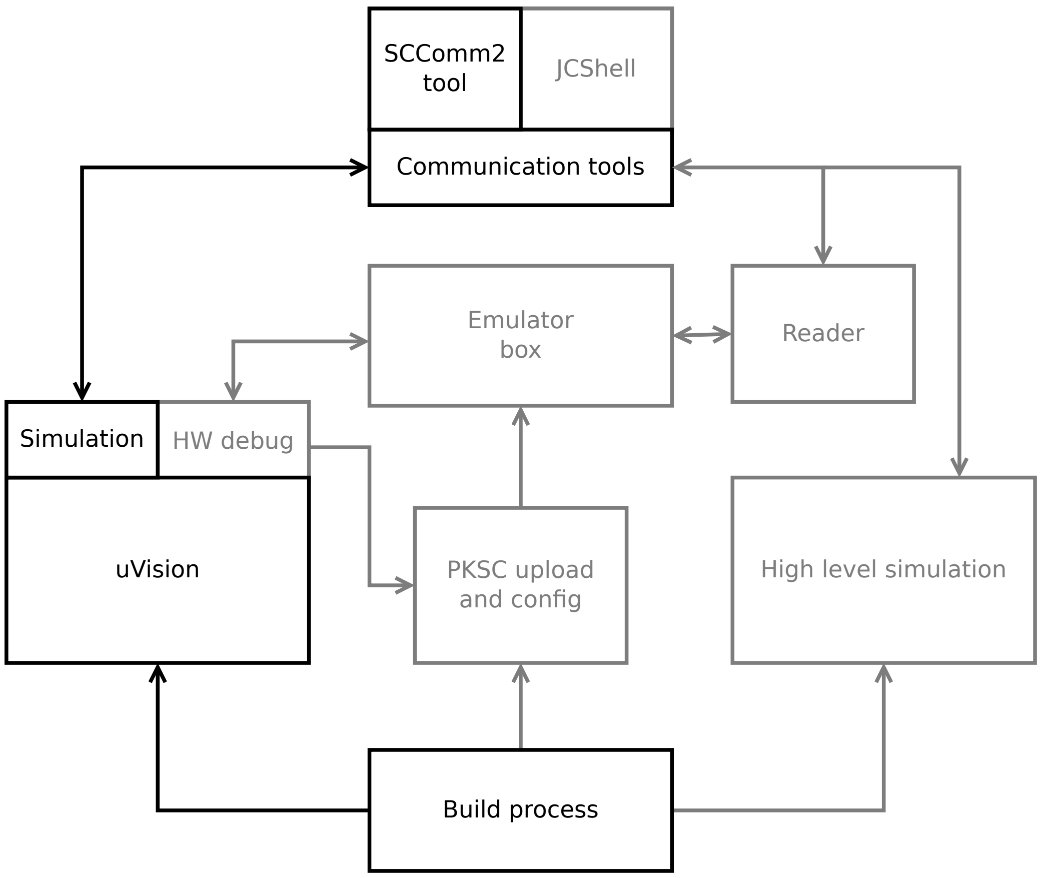

Figure 6.1: Testing environment

times into account it will be necessary to test on a real hardware platform. Figure 6.1 illustrates the overall testing environment emphasizing the parts particularly used within the course of this thesis.

6.2 Qualitative Evaluation

In the course of this master's thesis the feasibility and operational capability of Java Card patching for ROM mask based operating systems could be proven. Yet it is very important to note that the underlying platform hardware plays an essential role in establishing a high-performance patching mechanism. Without the respective environment elements described in Section 5.2.2 the mechanism would not be achievable in such a performant way. This is especially true for the data watch register mechanism and the exception or interrupt mechanism. The full support of these features in terms of the platform is vital. A pure software approach where every bytecode access address is checked by code in the virtual machine would be thinkable but is by far less practicable and decreases performance severely.

The most challenging task was to first establish the Java Card core patching mechanism on card from scratch and extend the patch code frame for supporting Java Card patches. That was one of the main tasks in the course of the thesis and turned out to be tricky in many ways.

It could be also worked out that the complexity in the application of Java Card patching clearly depends on the element to patch (patchable elements and subsequent analysis in Section 4.2). Writing patches may be a simple task but may also be really tough and close to infeasibility.

6.2.1 Java Card Patching Core Mechanism

When it comes to the Java Card patching core mechanism issues like usability for developer and performance are most important. It is important that the developer of a Java Card patch can find his way around in the patch code frame. From the patch developer point of view it is easy to insert and activate the patch once it is generated. Just a few marked locations in the Java Card code frame have to be touched, nothing else is required. The resulting patch image is then the readily filled Java Card code frame. In terms of performance the Java Card patch core mechanism is not introducing any significant delay and the memory usage is small (quantitative evaluation in Section 6.3).

6.2.2 Executable Bytecode Patching

Patching executable bytecode showed to be the most important and promising type of Java Card patching. With the presented approach it is possible to add, replace, or remove Java Card bytecode. The patch code generation process can be virtually always conducted by means of automatic generation or Java compilation and Java Card conversion, respectively. That is very comfortable even though it could be seen that special bytecodes can become necessary. That is for instance the case when the maximum size of the stack gets bigger just by altering code and assumptions of locality and relative referencing are made by the converter that do not hold for the patch at the patching area in EEPROM. A special bytecode for jumping out of the patch is very often required (special bytecodes in Section 5.5.1). Simple straightforward automatic generation cannot be applied. It remains a semi-manual task. That fact lies in the nature of the patch creation process and is hardly automatable. It is still essential that patch code creation remains a feasible task. Exactly this differs from case to case. An example in the form of a use case is given in Section 6.3.

In terms of performance regarding runtime and memory it is hard to give a general statement because this depends heavily on the specific case. Executable bytecode patching per se is not inducing much performance impact as it is simply an execution of bytecodes out of the patching area in EEPROM including some potential special bytecodes. The principal code quality of the patch heavily depends on the patch code developer.

6.2.3 Local Variable Patching

In Chapter 5 an accurate technical description of the local variable patching process was given. More precisely, it is about the emulated addition of a local variable by providing a special purpose memory area and the corresponding bytecodes for access. However in the

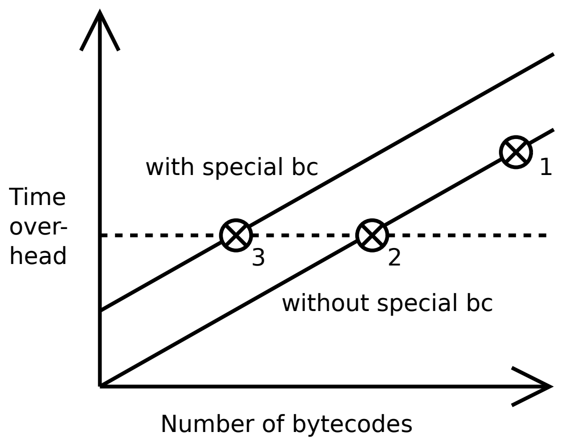

Figure 6.2: EEPROM access times and special bytecode overhead

course of this master's thesis it was not possible to implement the local variable patching on the platform because of the limited time frame.

It can be stated that creating this kind of patching and the corresponding core mechanism extension is very sophisticated and more difficult than the approach presented in Section 6.2.2. One reason is that patch creation is becoming a big problem because the compiler and converter are not aware of the special conditions. When adding local variables for the patch generation the resulting bytecode may be totally different than the original bytecode to be patched as variable placement on the Java Card stack may be altered. Many manual amendments are getting necessary, at least every local variable access has to be checked and the accesses to the newly introduced variables have to be exchanged by special bytecodes. The next issue is the potential parallel usage of the special memory area by different patches. This fact induces many problems and possible side effects but is yet a trade-off in terms of a practicable solution. Another point that is taking the focus away from local variable patching is that it can be bypassed by executable code patching in many cases. It is possible to avoid local variables to some extend by adding more code and nesting. Local variables are often used for better visibility and as temporary memory only. In that case they can be omitted.

6.2.4 EEPROM Access Times and Special Bytecode Overhead

The patch image and thus the contained Java Card patch executables are stored in EEPROM. As access times for ROM and EEPROM differ, the amount of executed patch bytecodes and thus the patch size influences the overall execution performance.

It could be concluded that a patch has a better performance the smaller it is because that limits the EEPROM read accesses. This statement is only partly true (compare below) as the special overhead of a patch has to be considered as well. A small patch having a return-from-patch special bytecode may be slower than a bigger patch without any need for that overhead. The principal task is to find the smallest possible patch without special bytecodes first. If a smaller one with special bytecodes can be found the induced overhead has to be exceeded by the gain through less EEPROM accesses. This situation is illustrated in Figure 6.2. In *point 1* we can see a first patch without special bytecodes, *point 2* shows the improved version and the smallest possible patch without special bytecodes. *Point 3* is a patch with the same overall time overhead but including special bytecodes (shown as additional offset). All potential patches with special bytecodes below point 3 are considered to be worth realized as the resulting time overhead is smaller as for any patch without special bytecodes. The impact of the the special bytecode offset also depends on the EEPROM access time overhead. When EEPROM access gets slower the relative influence of the special bytecode execution offset (executed in ROM) gets smaller as well.

6.3 Quantitative Evaluation

In the previous section it could be seen that Java Card patching works well and is besides unequally successful in different variants. The Java Card patching core mechanism was also evaluated quantitatively. In this section some measurements concerning overhead induced by the mechanism are given. General quantitative statements about applied Java Card patches are not expressive as they differ in the amount of code and their purpose from case to case. The topic of real-world use cases will then be covered in Section 6.4.

Many things around the overall patching mechanism are hard to measure in terms of quantitative values. Meaningful figures concerning patch creation are difficult to gain and real-world patch placement was not evaluated. Moreover many overall performance values depend on the single application and the patch code created by the patch code developer. The Java Card patching core mechanism though is always involved and can be subject to reasonable analysis.

6.3.1 Java Card Patching Core Mechanism Performance

The Java Card patching core mechanism is part of the execution for every applied patch on card and thus important with respect to performance. Every Java Card patch is based on the mechanism and inserted into it. It gets essential in every successful patch application and execution.

Memory Consumption

The Java Card patch mechanism and the patch code frame in particular induce some overhead in used memory area. As the patch code frame is existing in EEPROM once the placement was accomplished, it is important that it is memory saving.

Table 6.1 shows the size of the patch code frame. As first point in the listing the plain frame without any special Java Card amendments is shown. The plain Java Card patch environment and the amended combined patch environment then induce 33% extra costs in comparison to a plain code frame (for native patching only) but are very small in absolute numbers. However it has to be noted that the plain figures are hard to appraise and have to be seen related to an applied patch (more in Section 6.4).

Table 6.1: Patch code frame - memory consumption

Code frame	Size [B]	Rel. to Plain [%]
Plain native code frame	87	-
Code frame with JC extension	116	+33.33

Activation Time Consumption

In this section the measurement of the activation time for a Java Card patch is shown. The importance of time consumption in the activation sequence is rather low as this is only conducted at startup in the smart card runtime. Yet it is giving a first impression of range of time consumption for the patching mechanism. Generally, time consumption has to be seen in comparison to typical smart card overall applications as well. Ticketing applications are in the range of several 100 ms, e-government applications as e-passport checks take several seconds. Table 6.2 shows the overhead of the activation phase in absolute numbers.

Table 6.2: Java Card patching core mechanism - activation time consumption

Activation activity	Time [ns]
Patch activation	5000

Switching Time Consumption

An indicator that is even more important and interesting than memory and activation time consumption is the execution time overhead induced by the Java Card patching core mechanism. The overall switching time from common execution to actual patch code execution is the one interesting issue here. Table 6.3 shows the overhead in absolute numbers which can be considered as small in total both for starting the patch and returning from patch in case a return-from-patch special bytecode becomes necessary.

Yet the expressiveness of the absolute numbers is controversial. It has to be considered in the context of an applied patch. The impact on the SCOS and card applications heavily depends on the frequency of the patch triggering and execution as any execution implies the mechanism preparation steps. Furthermore it can be considered relative to the patch code execution time. In comparison to very small patch executable the overhead may be quite big, for bigger patch executables it is possibly minimal.

Table 6.3: Java Card patching core mechanism - switching time consumption

Switching activity	Time [ns]
Start patch	3000
Return from patch	1750

6.4 Java Card Patching Use Case

To get a comprehensive picture of the overall Java Card patching mechanism and a reference to praxis a typical use case scenario for a standard application is presented within this section. By means of this example a typical patching situation, the corresponding code generation process, and the performance results are detailed. The basic example code is out of the tutorial section of the Java Card Development Kit [Ora10a]. Because of technical reasons it was slightly amended but is original with regards to content.

In Code Snippet 6.1 a Java function for PIN verification on a typical ATM card is shown. A PIN is going through a verification chain and in the case of a wrong PIN or any non-conformity an exception is thrown and appropriate status words are returned (e.g. `SW_PIN_FAILED`).

Listing 6.1: Code example - PIN verification method

```java
/**
  * Handles Verify Pin APDU.
  * @param apdu APDU object
  */

private void processVerifyPIN(APDU apdu) {

  byte[] buffer = apdu.getBuffer();
  byte pinLength = buffer[ISO7816.OFFSET_LC];
  short count = apdu.setIncomingAndReceive();

  if (count < pinLength)
    ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

  byte pinType = buffer[ISO7816.OFFSET_P2];
  switch (pinType) {
    case MASTER_PIN:
      if (!masterPIN.check(buffer,ISO7816.OFFSET_CDATA, pinLength)){
        ISOException.throwIt((short)(SW_PIN_FAILED));
      }
      break;
  ...
}
```

In the scenario it is required to do a change in code and functionality as shown in the next code snippet. This may become necessary because the environment changes and other status words are expected or a used test suite was insufficient in terms of test coverage and the wrong status word response APDU was accepted erroneously.

In the original code only the status word `SW_PIN_FAILED` is returned. This is altered to the return of `SW_PIN_FAILED` plus the remaining tries of PIN verifications (card gets locked after defined number of tries) as shown in Code Snippet 6.2.

Listing 6.2: Code example - changes in source code

```
if (!masterPIN.check(buffer,ISO7816.OFFSET_CDATA, pinLength)){
  ISOException.throwIt((short)(SW_PIN_FAILED));
}

...changes to...

if (!masterPIN.check(buffer,ISO7816.OFFSET_CDATA, pinLength)){
  ISOException.throwIt((short)((SW_PIN_FAILED) +
                                  masterPIN.getTriesRemaining()));
}
```

For the patch code generation, automatic generation via Java compiler and Java Card converter is used. The source code is amended right within the usual development environment and Java Card bytecode is emerging in the end. When generating patch bytecode for application on card it is quite difficult to see the detailed differences between original and resulting code. Furthermore it is demanding to get a small patch size.

For this very example we can see that Java Card bytecode changes significantly on three positions shown by Code Snippet 6.3 and Code Snippet 6.4.

Listing 6.3: Code example - original Java Card bytecode

```
                  # block block 35,647-647, 0x81df71, 14
0xbe,             #   sload_5
0x75,             #   slookupswitch
                  #   0x81df73, 12
0x00,0x31,        #   relref 49, 0x81df72, 0x81dfa3
0x00,0x02,        #
0xff,0x81,        #
0x00,0x0d,        #   relref 13, 0x81df72, 0x81df7f
0xff,0x82,        #
0x00,0x1f,        #   relref 31, 0x81df72, 0x81df91
                  # block block 64,649-649, 0x81df7f, 10
0xad,             #   getfielda_this
0x02,             #   masterPIN, Lsystem/OwnerPIN ...
0x1a,             #   aload_2
0x08,             #   sconst_5
0x1f,             #   sload_3
                  #   L649, invokevirtual, system/OwnerPIN ...
0x8b,             #   invokevirtual
0x04,             #   call slot cnt OwnerPIN, ...
0x01,             #   public_slot
0x61,             #   ifne
0x22,             #   rel off 34, dst 0x81dfa9
                  # block block 77,651-651, 0x81df89, 8
0x11,             #   sspush
0x69,0xc0,        #   val 0x69c0
                  #   invokestatic, system/ISOException ...
0x8d,             #   invokestatic
0x08,0x5a,        #   ext method ref ISOException, throwIt ...
```

Listing 6.4: Code example - altered Java Card bytecode

```
                 # block block 35,647-647, 0x81df71, 14
0xbe,            #   sload_5
0x75,            #   slookupswitch <#<#CHANGES<#<#<
                 #   0x81df73, 12
0x00,0x37,       #   relref 55, 0x81df72, 0x81dfa9
0x00,0x02,       #
0xff,0x81,       #
0x00,0x0d,       #   relref 13, 0x81df72, 0x81df7f
0xff,0x82,       #
0x00,0x25,       #   relref 37, 0x81df72, 0x81df97
                 # block block 64,649-649, 0x81df7f, 10
0xad,            #   getfielda_this
0x02,            #   masterPIN, Lsystem/OwnerPIN ...
0x1a,            #   aload_2
0x08,            #   sconst_5
0x1f,            #   sload_3
                 #   L649, invokevirtual, system/OwnerPIN ...
0x8b,            #   invokevirtual
0x04,            #   call slot cnt OwnerPIN ...
0x01,            #   public_slot
0x61,            #   ifne
0x28,            #   rel off 40, dst 0x81dfaf <#<#CHANGE<#<#
                 # block block 77,651-651, 0x81df89, 14 <#<#CHANGES<#<#
0x11,            #   sspush
0x69,0xc0,       #   val 0x69c0
0xad,            #   getfielda_this
0x02,            #   masterPIN, Lsystem/OwnerPIN ...
                 #   invokevirtual, system/OwnerPIN, getTriesRemaining...
0x8b,            #   invokevirtual
0x01,            #   call slot cnt OwnerPIN, getTriesRemaining ...
0x02,            #   public_slot
0x41,            #   sadd
                 #   invokestatic, system/ISOException, throwIt ...
0x8d,            #   invokestatic
0x08,0x5a,       #   ext method ref ISOException, throwIt ...
```

On the basis of this example it can be seen that an alteration on one source code location can entail several changes in bytecode on different positions. In this case - next to the intended core changes - the code addition within the `switch` and the `if` statement leads to a shift of the different labels.

The first simple straight-forward approach would now be the patching of the entire `switch` statement. The patch would range from the first to the last bytecode difference between original and patched. This is fully functional and correct. This patch is referred to as *simple patch*.

However the patch size can be minimized. The original bytecode in ROM has fixed labels and actually no clue of the patch beforehand. As in the presented example only one line of source code or, more precisely, the parameter for the `throwIt` method is changing, there are no structural changes to the `switch` statement. Thus the resulting patch bytecode reflects only the directly changed source code and can be simply plugged in. The *small patch* is triggered right at the very core bytecode alteration position and is returning directly after it via special bytecode. The patch always contains the bytecode of the line where it is triggered as otherwise this bytecode could never be executed (exception is triggered before execution at every access). In addition to the core patch code only a minimal set of environment bytecodes is necessary. The return-from-patch special bytecode is included and points to the `invokestatic` call of the `throwIt` method. Code Snippet 6.5 shows the respective patch code.

Listing 6.5: Code example - small patch

```
// Triggering position at (original code):
#############################################

0x11,              #   sspush
0x69,0xc0,         #   val 0x69c0

// Executable patch code:
###########################

0x11,              #   sspush
0x69,0xc0,         #   val 0x69c0
0xad,              #   getfielda_this
0x02,              #   masterPIN, Lsystem/OwnerPIN ...
                   #   invokevirtual, system/OwnerPIN, getTriesRemaining...
0x8b,              #   invokevirtual
0x01,              #   call slot cnt OwnerPIN, getTriesRemaining ...
0x02,              #   public_slot
0x41,              #   sadd

// Return-from-patch special bytecode (4 byte)

// Return position at (original code):
#########################################

                   #   invokestatic, system/ISOException, throwIt ...
0x8d,              #   invokestatic
0x08,0x5a,         #   ext method ref ISOException, throwIt ...
```

Even though it seems as if the *small patch* above was ideal and minimal a very special case is faced in this example. It is assured that when a `throwIt` method is called there is no point of return. When the patch includes the `throwIt` method and always reaches that point no special bytecode for return-from-patch is needed. The method is executed out of ROM and the return-from-patch happens implicitly. Exactly this is achievable in the selected use case. The patch is referred to as *small advanced patch.* Code Snippet 6.6 shows the respective patch code.

Listing 6.6: Code example - small advanced patch

```
// Triggering position at (original code):
#############################################

0x11,              #   sspush
0x69,0xc0,         #   val 0x69c0
```

```
// Executable patch code:
##########################

0x11,            #   sspush
0x69,0xc0,       #   val 0x69c0
0xad,            #   getfielda_this
0x02,            #   masterPIN, Lsystem/OwnerPIN ...
                 #   invokevirtual, system/OwnerPIN, getTriesRemaining...
0x8b,            #   invokevirtual
0x01,            #   call slot cnt OwnerPIN, getTriesRemaining ...
0x02,            #   public_slot
0x41,            #   sadd
                 #   invokestatic, system/ISOException, throwIt ...
0x8d,            #   invokestatic
0x08,0x5a,       #   ext method ref ISOException, throwIt ...

// No explicit return from patch
##################################
```

6.4.1 Performance Measurements

The applied Java Card patches are also evaluated in terms of time and memory consumption. Table 6.4 shows the impact of the patch application on the `processVerifyPIN` method under the assumption that the method ends in the patched branch and the `throwIt` method. This is only conditionally expressive as the whole method is quite big in comparison to the change in code. The first line contains the measurement of the altered original code (patch-alike change in original code without any real patch), the second shows the naive simple patch, the third one shows the performance of the small patch, and the fourth the values for the small advanced patch. All changes are opposed to the unaltered original.

It can be seen that the patch has a relatively small impact on the overall method performance. Table 6.4 is showing the time measurement results. It can already be observed that the small patch is surprisingly a bit slower than the simple and the small advanced patch. More details are shown below.

Table 6.4: Java Card patch use case - Time consumption - Method

Change	Rel. to orig. [%]
Altered original	+1.13
Patch simple	+1.17
Patch small	+1.18
Patch small adv.	+1.17

To get a better impression of the patch performances and to explain the small differences in execution time mentioned above a relative comparison of the different patch versions is done in Table 6.5. The relative differences between the patch versions are very small but are interesting yet. The difference is existing because the simple as well as the small advanced patch do not use a return-from-patch special bytecode and are hence faster than the small patch version. The times are measured right from the bytecode where the simple patch is triggered. Under the used simulation environment EEPROM access times are disregarded (more in Section 6.1).

Table 6.5: Java Card patch use case - Time consumption - Patch to Patch

Change	Time diff. to patch simple [ns]	Rel. to patch simple. [%]
Patch small	+3000	+0.02
Patch small adv.	0	+0

The patch performance can also be evaluated with respect to the patch size in EEPROM. Table 6.6 shows the significant reduction of size and thus patch memory consumption in terms of the simple patch in comparison to the small patch and the small advanced patch.

Table 6.6: Java Card patch use case - Memory consumption - Patch to Patch

Change	Size [B]	Rel. to patch simple [%]
Patch simple	196	-
Patch small	129	-34.18
Patch small adv.	128	-34.69

6.5 Patching Project Costs Use Case

As seen in Chapter 1 patching is very often considered as an alternative to the creation of a new ROM mask. The option of a new ROM mask is not always given but especially in the production and evaluation process it is an option when a new error occurs. The main reason for patching within this process is cost reduction. By means of a generic example the advantages of patching to a new ROM mask are illustrated.

It is assumed that a new version of an SCOS on a smart card product is sent through the following production and evaluation chain comprising many different stages:

- *Development stage* comprises all steps to develop or advance an SCOS.
- *Test stage* comprises all tests at manufacturer side, from unit to system tests.
- *ROM masking stage* comprises all fabrication steps to mask an SCOS in ROM and bringing the smart card to a proper state. Sample cards are delivered.
- *Evaluation stage* is the testing phase at evaluation laboratories.
- *Release to customer stage* is the final stage where the product is already evaluated with positive results. It is released to the customer.
- *ROM code amendment stage* sets in when an error was detected and the decision is made to fix it in the original ROM code. Original ROM code is amended.

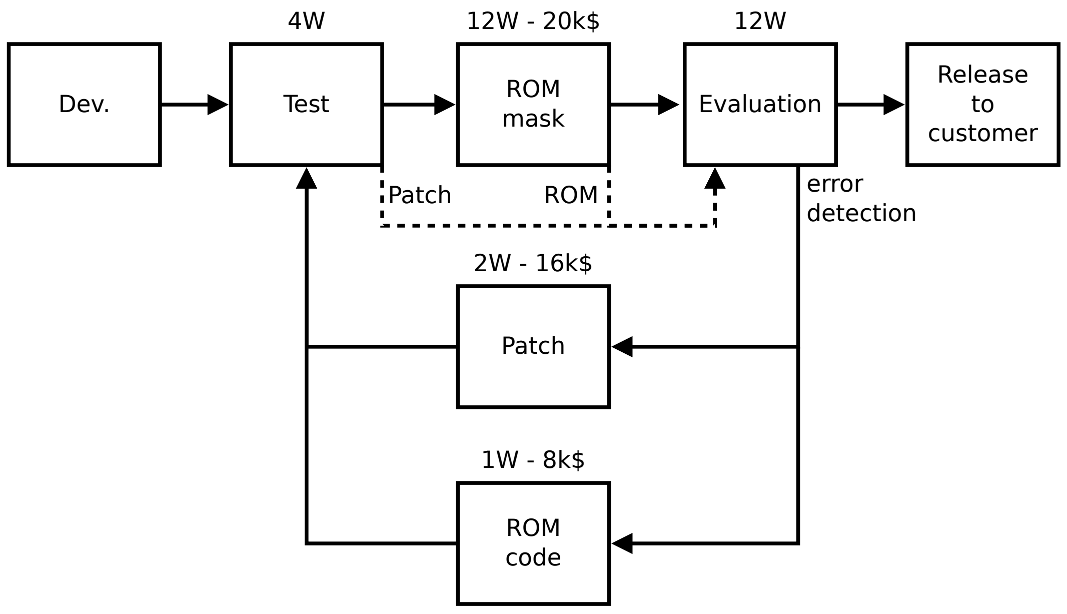

Figure 6.3: Project plan - patch versus new ROM mask

- *Patching stage* sets in when an error was detected and the decision is made to fix it via a patch. A patch is created.

Figure 6.3 illustrates a typical project plan with exemplary time and cost values for the items that differ in the new ROM mask and patching process. First of all an SCOS is newly created, improved or extended in the development stage. In the test stage the product is tested by the manufacturer itself. That leads to ROM mask fabrication and sample cards are delivered to the evaluation laboratory. Somewhere in the evaluation process an error is detected and an error correction process has to be initiated. The correction can be done by creating a patch or by ROM code amendments. Patch creation takes longer (up to two times) but a repetition of the ROM masking stage can be omitted. The required changes in the original ROM code are potentially faster to perform. However a new ROM masking process is necessary afterwards. Table 6.7 highlights the advantages of patching over ROM code amendments.

Table 6.7: Patch versus ROM code amendments - time and costs

	Time diff. to ROM [W]	Cost diff. to ROM [$]
Patching	-11	-12.000

Chapter 7

Conclusion and Future Work

The importance of a Java Card patching mechanism was clearly highlighted within this work. Absolute code correctness cannot be guaranteed in a current smart card operating system (SCOS) development. Thus the backdoor to error correction has to be left open by means of patching. It was already possible to patch errors for native code from assembler and C layer. The introduced mechanism extended patching to the Java layer and with it to the patching of Java Card bytecode and runtime areas of the Java Card environment.

An existing patch mechanism was extended comprehensively to support Java Card patching on the platform. To provide the underlying mechanisms for the hardware-supported solution, the SCOS (and with it the existing patch code frame) had to be made available first. The Java Card patching core mechanism development was divided into the creation of a Java Card code frame (for the Java Card patches to be plugged in) and the preparation of the SCOS patching functionalities. The generation of Java Card patches was another crucial topic. Many different issues had to be clarified and further amendments on the core mechanism were necessary to support different elements with various characteristics. Performance measurements were done on a hardware-related simulation regarding the mechanism itself and some patches out of a use case scenario.

The very special application domain of smart cards sets the general requirements for the Java Card patching mechanism. The fact of having a fixed SCOS in ROM was the initial trigger for the development of such a complex mechanism. The application of a Java Virtual Machine and Java as language for crucial parts of the SCOS were pushing for a solution as well.

In the course of this master's thesis it could be demonstrated that Java Card patching for ROM mask based SCOS is possible. The functional efficiency was shown on a working example. Value was set on the reasonable extension of the mechanism regarding the most probable and important types of patchable elements and related errors in Java based on expert knowledge and thorough analysis.

7.1 Future Work

The introduced Java Card patching mechanism can be considered as a mechanism for fixing an adequate set of possible faults and problems in the Java layer. The most important elements can be covered. The extension to further patchable elements is imaginable but may not become interesting until apparent urgent issues are detected. Especially the realization of the described treatment of local variables is an interesting issue. It is realistic that this gets essential in future as it addresses an important category of faults. As the presented approach is an emulation of local variables and entails some disadvantages as well, it might be of interest to develop a mechanism for the addition of local variables on the Java Card stack. The Java Card bytecode generation for the patch could be a potential target of automation in future. Right now it is described as a semi-automated procedure comprising compilation and conversion by means of the build process and many amendments by hand. Yet it will be very complicated and possibly not reasonable to automate this flow, as patch creation is no daily business. Likely, it does not pay off to go in this direction. The Java Card patching core mechanism itself could be further improved. More examinations in terms of performance are thinkable. As the mechanism introduces a fixed overhead in patch application it is crucial to make it as slim and fast as possible.

Bibliography

[AHKS93] H. Agrawal, J.R. Horgan, E.W. Krauser, and S.London. Incremental Regression Testing. In *Proceedings of the Conference on Software Maintenance*, pages 348–357, 1993.

[Ali91] S.R. Ali. Software Patching in the SPC Environment and its Impact on Switching System Reliability. In *IEEE Journal on Selected Areas in Communications*, pages 626–631, 1991.

[Ash10] Ashling. Ashling Development Tools. `http://www.ashling.com`, August 2010.

[Aya04] K.J. Ayala. *The 8051 microcontroller.* Thomson, Delmar Learning, 2004.

[BH00] B. Buck and J.K. Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14:317–329, November 2000.

[Car02] T. Carper. Smart Card Patch Manager, Januar 2002. United States Patent, US 6,338,435 B1.

[Car10] CardWerk. The ISO 7816 Smart Card Standard: Overview. `http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816.aspx`, August 2010.

[Che00] Z. Chen. *Java Card Technology for Smart Cards.* The Java Series, 2000.

[Com10] RS Components. RS Components - Electronic and Electrical Components. `http://uk.rs-online.com/web/`, September 2010.

[Dad05] J. Dadzie. Understanding Software Patching. *Queue*, 3:24–30, March 2005.

[Dai04] J. Daintith. *A dictionary of computing.* Oxford University Press, fifth edition, 2004.

[Dat09] A. Data. Patching Mechanism and Enhancements in SCOSTA Smart Card Operating System. Master's thesis, Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, 2009.

[EMV10] EMVCo. EMVCo Specification. `http://www.emvco.com/specifications.aspx`, October 2010.

[ET07] M. Ekman and H. Thane. Dynamic patching of embedded software. *Proceedings of the Real Time and Embedded Technology and Applications Symposium, 2007*, pages 337–346, 2007.

[GCC10] GCC. GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`, August 2010.

[Gie08] Giesenke & Devrient GmbH. Ausführung von Patches mittels eines Caches, August 2008. Europäische Patentanmeldung, EP 1 818 817 A2.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java^{TM} Language Specification, 3rd Edition.* Prentice Hall, 2005.

[Glo10] Global Platform. Global Platform Specification. `http://www.globalplatform.org/specifications.asp`, October 2010.

[Hac10] Hackingalert. Black hat or White hat! `http://www.hackingalert.com/hacking-articles/computer-hacking-explained.php`, July 2010.

[Hen01] M. Hendry. *Smart Card Security and Applications.* Artech House Inc, second edition, 2001.

[HID10] HID. Omnikey readers. `http://www.hidglobal.com`, August 2010.

[Hol01] M.R. Holiday. Patching Environment For Modifying A Java Virtual Machine And Method, March 2001. United States Patent, US 6,202,208 B1.

[HR04] I. Haddad and Ericsson Research. Towards Carrier Grade Linux Platforms. In *USENIX 2004 Annual Technical Conference, FREENIX Track*, pages 207–214, 2004.

[ISO10] ISO/IEC. ISO Standards. `http://www.iso.org/iso/iso_catalogue.htm`, August 2010.

[Kei10] Keil. Keil tools. `http://www.keil.com/`, August 2010.

[KLM09] C.M. Kirsch, L. Lopes, and E.R.B. Marques. Semantics-Preserving and Incremental Runtime Patching of Real-Time Programs. `http://www.dcc.fc.up.pt/~edrdo/papers/apres2008.pdf`, December 2009.

[Kru95] P. Kruchten. Architectural Blueprints—The 4+1 View, Model of Software Architecture. *IEEE Software*, 12:42–50, November 1995.

[LY99] T. Lindholm and Frank Yellin. *Java^{TM} Virtual Machine Specification, 2nd edition.* Addison-Wesley Longman Publishing Co., 1999.

[Man10] Manvir. Stored Program Control. `http://sensoria.in/TU/knowledge-pool/34-electronics-and-communication/46-stored-program-control.html`, July 2010.

[Meh10] J. Mehaffey. Runtime Application Patching for High Availability with Carrier-Grade Linux. `http://rtcmagazine.com/articles/view/100182`, July 2010. Montavista Software.

[Mic10a] Microsoft. Visual C++ Developer Center. `http://msdn.microsoft.com/en-us/visualc/default.aspx`, August 2010.

[Mic10b] SCM Microsystems. SCM readers. `http://www.scmmicro.com/`, August 2010.

[Nat02] National Informatics Centre, Ministry of Communication and Information Technology Government of India, Indian Institute of Technology Kanpur. *Specifications for the Smart-Card Operating System for Transport Applications (SCOSTA)*, March 2002.

[Ora03] Oracle Technology Network. *Virtual Machine Specification, Java Card Platform, Version 2.2.1*, October 2003.

[Ora10a] Oracle. Java Card Development Kit 2.2.2. `http://www.oracle.com/technetwork/java/javacard/downloads/index.html`, August 2010.

[Ora10b] Oracle Technology Network. Java Card Technology. `http://www.oracle.com/technetwork/java/javacard/overview/overview-jsp-135353.html`, September 2010.

[ORH02] A. Orso, A. Rao, and M.J. Harrold. A Technique for Dynamic Updating of Java Software. In *Proceedings of the International Conference on Software Maintenance (ICSM'02) table of contents*, page 649, 2002.

[ptr10] UNIX Manual Page: man 2 ptrace. `http://www.cl.cam.ac.uk/cgi-bin/manpage?2+ptrace`, July 2010.

[RE03] W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley & Sons Ltd, third edition, 2003.

[Sem10] NXP Semiconductors. Pegoda readers. `http://www.nxp.com/pip/066120.html`, August 2010.

[Smi10] Smithsonian. ROM, EPROM, and EEPROM Technology. `http://smithsonianchips.si.edu/ice/cd/MEMORY97/SEC09.PDF`, September 2010.

[Tec10] TechTerms.com. Virtual Memory. `http://www.techterms.com/definition/virtualmemory`, August 2010.

[Zho10] Y. Zhou. Definition Of Various Cache Algorithms. `http://www.usenix.org/events/usenix01/full_papers/zhou/zhou_html/node3.html`, July 2010.